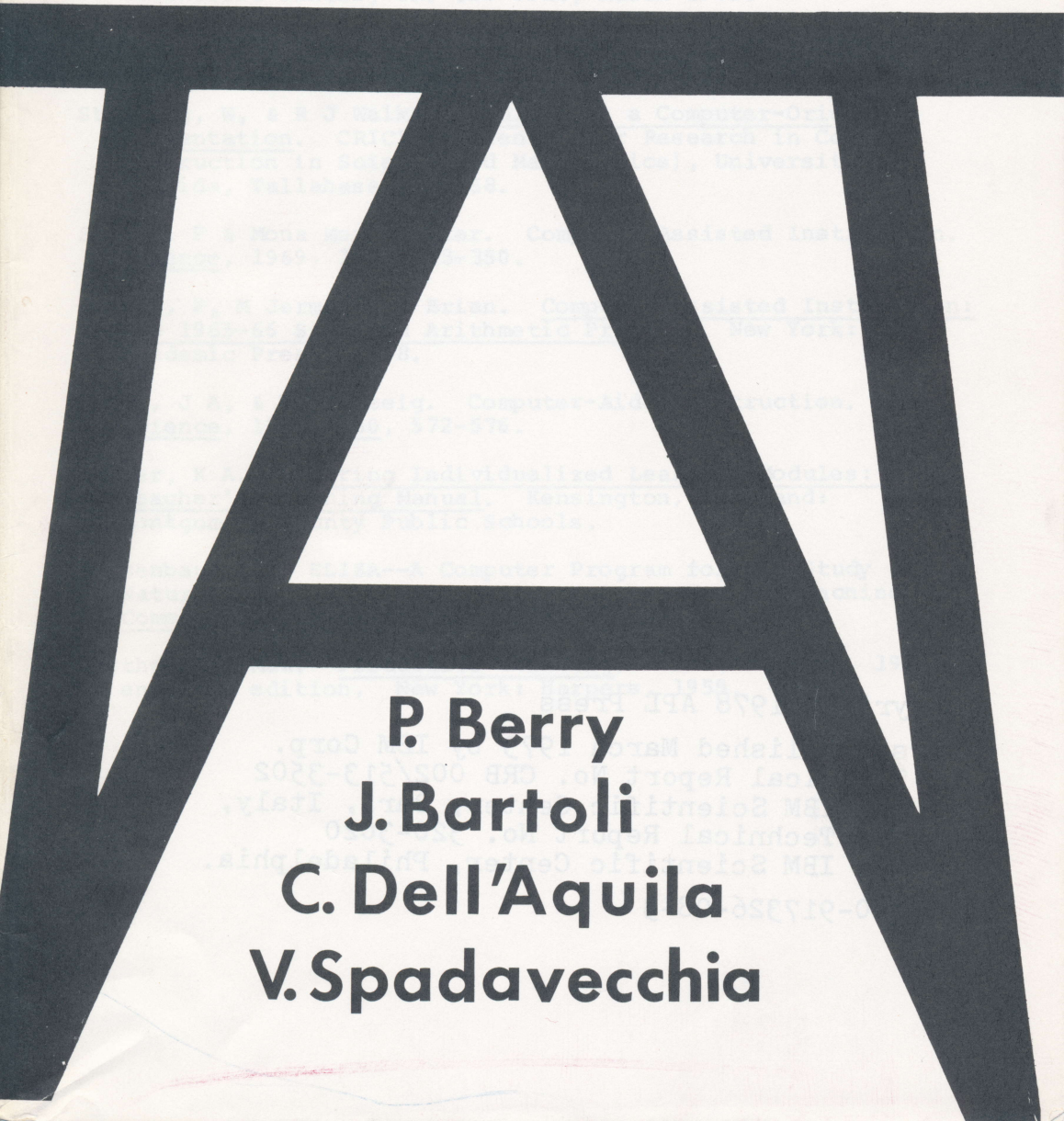
APL and INSIGHT



P. Berry
J. Bartoli
C. Dell'Aquila
V. Spadavecchia

First published March 1973 by IBM Corp.
as Technical Report No. CRB 002/513-3502
of the IBM Scientific Center, Bari, Italy,
and as Technical Report No. 320-3020
of the IBM Scientific Center, Philadelphia.

ISBN 0-917326-08-3

Contents

1. APL AND THE STRATEGY OF TEACHING

Introduction

This report is based upon three central ideas:

that key concepts in various disciplines may be represented by functions;

that a language such as APL permits a readable, formal definition of a function and a means of executing it and thereby accumulating the experience necessary to understand it; and

that it is possible (but, unfortunately, not usual) to write computer programs so that they correspond directly to the functional concepts of a discipline.

Efforts to use the computer in education have in the past been focussed primarily on measuring educational achievement by using its data-processing powers, on its ability to present and manage programmed instruction, or on the machine itself (as the subject of study). A new role for the computer emerges when the discipline of programming is used to represent explicitly a topic's structure, and thus to provide a basis for insight.

To say that a student has gained insight into a topic means that he has come to see an underlying structure, and that, by referring to that structure, he is able to state and apply rules which explain or predict. In particular, he is able to abstract the rules from their context, and generalize them to new situations. One requirement of an educational process is that it provide the student language in which to describe his disciplines and in which to think about them.

Insight and the Concept of Function

The fundamental ideas of a discipline depend upon two aspects of the way we choose to represent it:

1. Data: those attributes that we choose to note and measure thereby become the facts by which knowledge of an event is encoded, and in terms of which we think about it.

2. Functions: the transformations from one representation of data to another, or the relations between different sets of data.

The choice of functions by which data are treated influences the selection of facts worth observing. Taken together, a collection of functions imposes a structure on the understanding of a topic. For some time, Falkoff and Iverson have argued that a system of programs constitutes a framework for a discipline. (See, for instance, Iverson, 1969, p. 53.) Iverson has also suggested (1972, pp 230-242) that the languages of science and mathematics resemble informal speech in that each contains nouns (to refer to data or observations), and verbs (to describe actions or transformations). The functions (formulas) which describe how data are related and transformed might be thought of as active, transitive verbs. It is often possible to choose the definitions of functions so that they correspond directly to terms used in less formal discussions of the same topic, and thereby achieve a parallel between the structuring of a topic provided by natural-language description and the formal structure of a system of data and functions.

Typically, a student comes to understand a function from three kinds of experience:

1. Knowing how the function is defined; i.e. how its definition is composed from previously established or more primitive concepts.

2. Experiencing its properties. The intense, intuitive familiarity which accompanies understanding is gradually built up in two ways:

   a. Accumulated experience of the way the function behaves in a variety of contexts.

   b. Formal analysis of the function's definition.

3. Seeing a particular function in its place as part of a larger system of functions. This is not a property of a single function, but of a more general system of description.

If a topic is well understood and its description well organized, a function useful in the description of one phenomenon can usually be used again in others. The creation of a unified and consistent system of definitions is, of course, what scientists are continually attempting. In similar fashion, teachers or authors can select the functions by which they develop a topic so that these form a coherent system.

Those functions which convey the greatest insight are those which can be transposed or generalized from one setting to another. Power comes not only from ad hoc skill at handling particular cases, but from knowledge of a principle that will permit the same skill to be applied in diverse contexts. Certain simple functions (for example, those of exponential growth or decay, or the trigonometric functions) occur in so many trades and sciences that we cease to regard them as part of any particular specialty. The functions of arithmetic are so universal that it frequently comes as a surprise to students to discover that addition and multiplication, too, are functions.

Poets or artists sometimes decry the precision that the sciences espouse; they claim to prefer instead the many-layered ambiguity that literature permits. They miss the point that the generality that in art is achieved by ambiguity is equally an aim of science. Scientists, like poets, seek to state a function so generally that it applies to things which once seemed unrelated, and thereby gain some insight into all of them.

In his studies of productive thinking among school children, Wertheimer (1954, 1959) stressed repeatedly the need to find a more general statement of a rule, so that functions learned in one context could be extended to new situations. His primary emphasis was upon the process of thought by which students may break out of misconceptions or poorly-stated rules, but his conclusion is directly applicable here: productive thinking results from organizing what we learn into broadly stated functions, whose implications and applications can be extensively explored.

Laws of physics, procedures of accounting, engineers' tables, models of the economy, of the population, or of the structure of attitudes -- all these, in disparate fields of endeavor, can be understood as functions. Some are functions by which observations of nature are summarized and extrapolated, and others are conventions by which commerce is organized. When a computer is used in this way, clear analysis of the functions to be performed benefits programming quite as much as pedagogy. Too frequently computing has been regarded as a way of getting answers, rather than as a way of stating and using methods. Here is the central point:

> We may choose to define a function, for the computer just as for the blackboard, in a way that not only gets the work done, but which reflects or illuminates the line of reasoning behind its definition.

The notion of function is surprisingly familiar. It is one of the few words whose popular usage is quite consistent with its more formal definition. People use phrases like "It's a function of time" in a way that they need not unlearn when they study mathematics. Conversely, a number of mathematical functions are extremely familiar, and are widely taught in elementary school (although they are there referred to as "operators"). Systems analysts and engineers also identify functions of subsystems. They represent those units as boxes or "functional units" which work on an input provided at one side, and produce an output at the other. This corresponds directly to the more mathematical concept of function, under which we speak of the result produced when a function is applied to an argument. For instance, in an expression such as y=cos x, the argument x is to be transformed by the function cos to produce a result y.

Insight and the Notion of Arrays

Handling arrays is commonplace in data processing; frequently the prime motive for using a computer at all is the need to process an array of data. But for the student seeking to understand a function, arrays have another significance. In building up his grasp of the functions used to make sense of the world of data, he can employ two strategies:

1. Given a rule of nature (that is, the definition of a function), examine its consequences.

2. Given separate facts or observations, deduce a rule (that is, the definition of a function) which approximates them.

A student can explore the behavior of a function by applying it to an array whose members vary systematically, and looking for patterns produced in the arrays of results. For instance, suppose that F is a function such as $\frac{1}{2}gt^2$, expressed in APL as `0.5×G×T*2`. Suppose X is an array of possible values for T chosen so that each differs from the next by 1, for instance `X←0,⍳5`.

X	F X
0	0
1	16
2	64
3	144
4	256
5	400

The student can examine the array *F X*, and ask whether its elements are in the same order, or also differ by a constant, and so on. Suppose *D* is a function which finds the difference between each element of an array and its predecessor. Applied to the values of *X* and *F X* shown above, *D X* gives 1 1 1 1 1, while *D F X* gives 16 48 80 112 144. The student may be led to the discovery that for some functions, all the elements of *D F X* are the same. For other functions, a constant difference is found only after several repeated applications of the difference function (such as *D D D F X*), or never. (This approach to examining the properties of functions is heavily exploited in Iverson, 1972.)

Similarly, when students are attempting to deduce the functions by which nature orders events, they may best start from tables in which one or another attribute is arranged in a systematic way. In Part 2 and in a separate report by Bartoli et al. (1972) this method is applied to some observations in elementary physics.

Whether the student starts with a function and generates results, or starts with observations and attempts to supply a function, the task is to discern within an array a pattern formed by systematically ordering the observations, or by systematically exploring a function's results. This, of course, is the reason that graphs are so widely employed in all the sciences: their two axes show the various points or cases arranged in order with respect to those two dimensions.

Insight arises not from particular instances but from the patterns which emerge from treatment of an array. Thinking in terms of arrays is not merely a notational convenience, or a way of saving labor (although it provides both in abundance), but an important aspect of organized thought.

Educational Relevance of Some Features of APL

In talks about their joint work on the development of APL, Iverson and Falkoff have stressed clear communication. APL has been intended as a language that permits clear and concise statement of procedures for the benefit of thought as well as for execution by machine. The educational relevance of the language follows from its salient characteristics as a programming language:

1. A simple syntax consistently applied to all functions.

2. Simple rules for applying functions to arrays, so that arrays may be treated as wholes.

3. A rich set of primitive functions, including many from mathematics and logic, and new ones for manipulating arrays.

4. The ability of each user to extend the language indefinitely by defining new functions in terms of those already defined (including functions that the user himself defined earlier).

In each field or discipline, the user may in effect extend the language to embrace the functions appropriate to it. The user may "pyramid" definitions, using those from each level as terms in the definition of those at the next.

Since APL permits arrays to be treated as wholes, a function whose definition applies to a single case (i.e. a scalar) can often be expressed so that it applies equally well to an array. This property greatly simplifies many expressions. APL in this regard resembles and extends matrix algebra. Operations on matrices (i.e. tables) are central to thinking about many topics, including a very large portion of business data-processing.

The APL\360 system (complementing the APL language) keeps to a minimum the steps that a user need go through to carry out a calculation. When a user offers the definition of a new function, as far as possible he has only to write its mathematical kernel. To have an expression executed, he has only to state it. This makes it possible for a student to make extensive use of the computer while having to learn little that is not directly relevant to his discipline of study, and nothing at all about the computer's internal mechanisms.

In summary, the attributes of APL valuable to education are the central place it gives to the notion of function, the formal consistency with which functions are treated, and the possibility of choosing functions so that they not only perform calculations, but correspond pedagogically to the concepts out of which the student builds his understanding.

The Art of Constructing Programs To Represent Functions for Teaching

The observation that "function" is a fundamental concept in many disciplines is by no means new, and neither is the idea of using the computer to provide immediate concrete experience of functions at work. Nevertheless, to make good use of these approaches in education requires a style and emphasis in

programming different from common usage in data processing. We are advocating a style of programming in which the programmer's responsibility is first to find the fundamental ideas, and then to provide a set of functions directly corresponding to those ideas.

What would constitute a "functional approach" to programming? First, in treating any particular topic, the aim must be to write a clear and concise statement of the principles involved. The relevant concepts upon which the function is based must be identified, and a definition provided for each. Each of these definitions should be a function in the formal sense that it takes an argument and returns a result, and thus may form part of a compound expression. When this is done with care, the definition provided for a particular concept will frequently reduce to a simple expression built from the relevant component functions. The definition will be many-layered: at any particular level, it will be a simple statement made up of units which in turn have their own definitions, which can either be taken for granted or themselves examined in more detail if the student needs.

Not only should the definition make use of relevant definitions established earlier, but it should be written so that it in turn may become a component of subsequent definitions. This means that the central definition should be carefully separated from ad hoc embellishments relevant only to a particular case. For instance, if special provision is to be made to facilitate use of the function by providing conversational input or by formatting tables of output, those ancillary services should be provided by a separate function, external to the function that embodies the main concept.

The author of a set of definitions for a family of functions to treat a particular topic is in effect designing a special language for that discipline, whose special terms are defined in APL and which operates under the general syntax of APL. His special language can and should reflect the language and concepts of the discipline he serves. It may also enrich them by extending the ways in which they can be examined and applied.

Identifying the Functions in a Sample Problem

The correlation coefficient is used to measure the extent of (linear) association between two sets of data. A computational formula commonly given for the correlation between two sets X and Y is

$$\frac{N\Sigma XY-(\Sigma X)(\Sigma Y)}{\sqrt{(N\Sigma X^2-(\Sigma X)^2)(N\Sigma Y^2-(\Sigma Y)^2)}}$$

in which X and Y are vectors of length N. This formula could be directly transliterated to APL. But in explaining correlation, a statistician is more likely to describe it as the cross-product of normalized scores. The cross-product is the sum of the products of one set of scores with the corresponding members of another set. If the data are treated as a 2-column matrix X (rather than as two separate vectors) correlation could be defined as

```
    ∇ Z←CORRELATION X
[1]   Z←CROSSPRODUCT NORMALIZED X
    ∇
```

To normalize the matrix X means to rescale the values within each column so that their mean is zero and the sum of their squares is one. The crossproduct is defined thus:

```
    ∇ Z←CROSSPRODUCT X
[1]   Z←(⍉X)+.×X
    ∇
```

and normalization is achieved this way:

```
    ∇ Z←NORMALIZED X
[1]   X←X - (⍴X)⍴MEAN X
[2]   Z←X ÷ (⍴X)⍴SQRT +⌿X*2
    ∇
```

which in turn makes use of functions for the mean and for the square root, whose definitions are obvious.

The *CORRELATION* function thus defined produces results identical to those found by the formula given first. But the APL definition analyzes the task into subtasks that make sense in statistics, and will find frequent use in related applications. Moreover, the function *CORRELATION* will produce a matrix of all the intercorrelations for any number of columns of observations.

Some Points of Style and Strategy In Programming for Insight

From the experience of devising sets of functions to treat various topics, one gradually evolves what amounts to a style of programming, perhaps un-learning some established habits on the way. In what follows we list several of these stylistic issues, including examples to illustrate the points being made.

1. Functions as Modules. A topic is best treated by a family of simple functions designed in modular units or "building blocks," rather than by a single monolithic program.

Modularity was evident in the preceding example concerning correlation. Another example follows. A program written for a class in a business school calculated how well various simple functions could be used to relate one economic indicator to the forecasting of another. An initial version of this program, written in monolithic rather than modular style, consisted of about thirty APL statements. The program included provision for interaction with the user to ask him which data he wanted, what weights were to be applied to smooth the observed data, what degree polynomial was to be fitted, and so on. After these were specified, tables of results were printed, and then the program asked the user how he would care to modify the parameters for another iteration. To illustrate some advantages of modularity, we undertook to divide this single function into subfunctions which would make sense within the framework of the business application.

Sometimes the quest for an appropriate set of building blocks is a subtle and difficult task. To get some impression of concepts already established in the field, and hence likely to be useful modules, one should examine a natural-language summary of the program's goals. The choice of terms for a natural-language explanation probably reflects the author's professional experience and insight into the way the task is organized. We noted all verbs occurring in a brief English description of the program, and found the following:

smooth	(a time-series of observations)
fit	(a polynomial by least squares)
evaluate	(a polynomial)
compare	(observed data with the fitted curve, by finding the sum of the squares of the differences between them)

The description also included the following nouns:

weights	(used in smoothing the time-series data)
degree	(of the polynomial to be fitted)
commodity	(whose production is being predicted)
indicator	(the economic indicator used to forecast the behavior of the commodity)

In our example, the criterion was corn yields per acre, and the predictor was capital invested in agricultural equipment.

The procedure called for smoothed data --that is, data in which each observation is modified by those preceding it in order to "smooth" out local variations. We first provided a function *SMOOTH*, taking two arguments: on the right, the data to be smoothed, and on the left a set of weights indicating how much the preceding observations contribute to a smoothed point. For instance, to smooth data *X* so that each element of the result combines five of the raw entries in an exponentially decreasing way, the student might set *W*←*0,-⍳5, and then obtain his smoothed data by

```
W SMOOTH X
```

If the process of smoothing itself is not at issue, this simply becomes the input to the function that does the curve-fitting. But if the student wishes to examine the consequences of smoothing, he may plot or tabulate his raw data against the results of smoothing with various weighting schemes. He may also examine the definition itself; a possible one follows:

```
    ∇ Z←WEIGHTS SMOOTH DATA;D
[1]   WEIGHTS←⌽WEIGHTS÷+/WEIGHTS
[2]   DATA←((⍴WEIGHTS),⍴DATA)⍴DATA
[3]   DATA←(0,⍳D←¯1+⍴WEIGHTS)⌽DATA
[4]   Z←D↓WEIGHTS+.×DATA
    ∇
```

The main concern of the exercise was the fitting of a polynomial to relate one set of data to another. A polynomial is a linear combination of the successive powers of a variable; the best-fitting coefficients (by the criterion of least squares) are found by matrix division. The right argument of *FIT* is a two-column matrix in which the second column contains the observed data, which are to be fitted by a polynomial in the variable shown in the first column. The result is the set of coefficients for the best-fitting polymonial, in ascending order of powers.

```
    ∇ C←DEG FIT X
[1]   C←X[;2] ⌹ X[;1]∘.*0,⍳DEG
    ∇
```

A polynomial whose coefficients are stated in that form can be evaluated by a function such as this:

```
    ∇ Z←X EVALUATE C
[1]   Z←(X∘.*¯1+⍳⍴C)+.×C
    ∇
```

Thus if the student needs to fit, for instance, a third-degree polynomial in order to approximate corn yields per acre by capitalization of farm machinery, he would find the coefficients of that polynomial by an expression such as

```
      3 FIT CAPITAL AND CORN
```

in which *AND* is a simple function which joins two arrays as the successive columns of a matrix. Taking the fitted polynomial, the student could evaluate it for the observed data on capitalization of farm machinery, and see how well this projection fitted his observations of corn yields. To evaluate the polynomial, he would execute

```
      CAPITAL EVALUATE C
```

Or he could combine all these steps in a phrase such as

```
      CAPITAL EVALUATE 3 FIT CAPITAL AND CORN
```

Finally, in order to measure the goodness of fit of this projection against the observations, the student should compare the observed with the projected data by computing the residual sum of squares:

```
    ∇ Z←A RESIDUAL B
[1]   Z←+/(A-B)*2
    ∇
```

To understand the significance of that residual will require them to study the analysis of variance, which is beyond the scope of this example.

The definitions shown above are shorter than the monolithic program that suggested this exercise. What has been left out? These functions make no provision for checking that they are used in a reasonable way, no provision for conversing with the user or printing a summary of his results, and no guidance for how he might adjust his previous attempt in order to achieve a

better approximation. On the other hand, they have a direct correspondence with the concepts of the discipline, and the student is free to construct his own expressions with them, including (should he wish to) embedding them in a conversational program.

2. Formal Structure of Functions. In general, a function should take explicit argument(s) and return an explicit result, without global arguments or global results.

The ability to use defined functions in new expressions depends upon having explicit results which may be passed directly to the next function as its argument. This ability is lost if the student uses what might at first appear to be a simple form of function definition, without formal arguments or results, or if the functions, when applied to a new use, cannot be restrained from printing messages about what they are doing.

A beginning programmer might conceivably write a pair of programs such as the following:

```
    ∇ STATISTICS
[1]   N←⍴X ;' OBSERVATIONS'
[2]   'MEAN '; M←(+/X)÷N
[3]   VARIANCE
    ∇

    ∇ VARIANCE
[1]   'VARIANCE '; V←(+/(X-M)*2)÷N
    ∇
```

Executing *STATISTICS* produces a labelled report of the mean and variance of a vector of observations called *X*. Or the user could execute *VARIANCE* and receive a report only on the variance. But *VARIANCE* as here defined can be used neither to assign a result, such as *V ← VARIANCE*, nor as a building-block in some other function such as *SD ← SQRT VARIANCE X* because it has no explicit result. Moreover it depends upon having *X*, *M*, and *N* set up beforehand, and prints the message *'VARIANCE'* whether or not a message is called for in its new setting.

A major emphasis of systems analysts is to clarify how the various components of a system are related by enumerating the inputs and outputs of each subsystem. That task is greatly facilitated if each function can be defined so that it takes an explicit argument and returns an explicit result, and neither uses global variables as implicit arguments, nor specifies variables other than its result. Adhering to such a

discipline makes clear two questions whose neglect can cause much confusion: (1) What does this function depend on? (2) What does this function do?

3. For Readability, Phrases of Sub-Functions. Definitions consisting of phrases of functions can make a definition much more comprehensible than a long sequence of steps, even when those steps are themselves very simple.

The notion that programming is fundamentally the art of writing a sequence of statements, and therefore that simple programs are those built of simple steps, is one that dies hard. Consider the following contrasting sets of definitions:

```
     ∇ Z←P1 N
[1]    Z←0
[2]    I←÷N
[3]    X1←0
[4]    Y1←1
[5]    X2←X1+I
[6]    Y2←(1-X2*2)*0.5
[7]    D←(((X1-X2)*2)+(Y1-Y2)*2)*0.5
[8]    Z←Z+D
[9]    X1←X2
[10]   Y1←Y2
[11]   X2←X1+I
[12]   →(X2≤1)/4
[13]   Z←2×Z
     ∇

     ∇ Z←P2 N
[1]    Z← 2× +/DISTANCE X,[1.5] CIRCALT X←(0,⍳N)÷N
     ∇

     ∇ Y←CIRCALT X
[1]    Y←(1-X*2)*0.5
     ∇

     ∇ Z←DISTANCE X
[1]    Z←SQRT ¯1↓+/(X-1⊖X)*2
     ∇
```

Despite the difference in their appearance, the functions *P1* and *P2* perform the same calculation by the same method. That is, each estimates pi by summing the lengths of a series of chords in a unit circle. The end-points of each chord are found by dividing a radius into *N* equal parts, and then calculating the height of the circle above each. Distances between adjacent points are found by applying the Pythagorean

theorem to their coordinates. Indeed, *DISTANCE* could itself be defined as a phrase of sub-functions:

```
    ∇ Z←DISTANCE X
[1]   Z←PYTH D X
    ∇
```

in which *D* is the difference between one element of an array and its predecessor, and *PYTH* is the usual Pythagorean formula *SQRT +/X*2*.

Since pi equals the circumference of half a circle, the result based upon a single quadrant must be doubled. The definition *P2* corresponds rather directly to a natural-language description, such as "Twice the sum of the distances between the points on a unit circle above *N* evenly spaced divisions of the radius." As a by-product, the intermediate functions *DISTANCE* and *CIRCALT* are defined, and may have other uses in coordinate geometry. (Note also that *CIRCALT* could be defined simply as *○X*.)

4. General Definitions. Where possible, choose a definition so that a function has maximum generality.

A function is often first defined in response to a specific need, for which a specific definition would be adequate. But the function will be more generally useful if its definition is written in a more general way. For instance, suppose an application refers to the mean of a vector. A definition valid for vector arguments would be:

```
    ∇ Z←MEAN X
[1]   Z←(+/X) ÷ ⍴X
    ∇
```

What happens if this definition is applied to arrays other than vectors? According to the definition shown, the mean of a scalar would be empty. Perhaps that would be appropriate --or perhaps it would be more convenient to say that the mean of a scalar is the scalar itself. What about the mean of a matrix, or of a multi-dimensional array? In many applications it is useful to regard *MEAN* as a sort of reduction, collapsing one dimension of an array and returning a result for each of those that remain:

```
    ∇ Z←MEAN X
[1]   Z←(+⌿X) ÷ 1↑(⍴X),1
    ∇
```

Applied to a matrix, this revised definition returns a vector of means, one for each column. Applied to a three-dimensional array, it returns a matrix of means, one for each row-and-column. Applied to a vector, it retains the accustomed meaning.

Generalizing an iterative process is more tricky, since a test which appropriately controls work on a single case may not be directly applicable to an array. New and useful applications are often discovered by attempting to make the first ad hoc definition more general. Suppose we start with the following definition for the greatest common divisor (GCD) of two numbers *X* and *Y* by Euclid's algorithm:

```
    ∇ Z←X GCD Y
[1]   Z←X
[2]   X←X|Y
[3]   Y←Z
[4]   →(X≠0)/1
    ∇
```

As written, this definition can only be applied to a pair. Can we generalize it to find the GCD not of a pair *X* and *Y* but of a set *X* having any number of members?

Since the greatest common divisor cannot be greater than the smallest member, we can start by taking the minimum of the set. Then we can eliminate from the set any members which are even multiples of the minimum. Those that are not eliminated are replaced by their residues modulo the minimum. Repeat until this eliminates all but one, which will be the greatest common demonimator.

```
    ∇ Z←EUCLID X
[1]   Z←⌊/X
[2]   X←Z,(0≠Z|X)/Z|X
[3]   →(1<⍴X)/1
    ∇

      EUCLID 36 63 27
9
```

5. Alternate Definitions. Equivalent functions can be written in many different ways. Exploring alternative ways of achieving the same result can be most rewarding.

Returning to the preceding example, GCD, we might seek a different algorithm and consider its consequences. The greatest common denominator can also be found by analyzing the

prime factors of the set *X*. Suppose we have a function *PF* which, when applied to a single integer, returns a list of the powers to which the various primes must be raised in order to reach that number. Thus:

```
      PF 63
0  2  0  1
```

This indicates that among the factors of 63, the first prime (that is, 2) is not represented, the second prime (that is, 3) is represented twice, the third prime is not represented, and the fourth prime is represented once. This is a number representation scheme (based on a product-of-powers-of-primes rather than the more familiar number systems based on sum-of-products-of-powers-of-a-base). Suppose also that the function *PF* has been generalized so that, when applied to a vector of numbers, it produces a table (in the same form as the table produced by the representation function ⍴) with a column for each number being represented, and a row for each prime:

```
      PF 63 15
 0  0
 2  1
 0  1
 1  0
```

The function *PF* has an inverse function *N* which converts these factorial representations back into numerical values by an expression such as *PRIMES*×.**FACTORS*. Then the GCD is defined thus:

```
    ∇ Z←GCD X
[1]   Z←N ⌊/PF X
    ∇
```

The interesting thing about this definition is that it leads to a parallel definition for the lowest common multiple (LCM):

```
    ∇ Z←LCM X
[1]   Z←N ⌈/PF X
    ∇
```

Yet another approach to the GCD could be obtained by taking its name as a description of the procedure, i.e. the greatest of those divisors that are common to all the members of a set:

```
    ∇ Z←GCDI X;I
[1]   I←⍳⌊/X
[2]   Z←⌈/(∧/0=I∘.|X)/I
    ∇
```

It would even be possible although probably not useful to define *GCD* by an expression such as *GREATEST COMMON DIVISOR X* in which *GREATEST* involved ⌈/, *COMMON* involved ∧/, and *DIVISOR* returned an array of all divisors of *X*.

6. Introducing New Concepts by Generalizing Earlier Ones.
Returning to simple functions introduced at one level of the curriculum and replacing them with more sophisticated generalizations offers an integrating link through different levels of the curriculum.

During an elementary course students might be introduced to successive approximations. For instance, they might write a function to extract square roots by repeatedly refining an initial guess until an adequate approximation is found. Such a function might begin with the arbitrary guess that the square root of any number is 1, and then repeatedly revise that last trial value by subtracting from it one-half the relative difference between its square and the target.

```
    ∇ Z←SQRT X
[1]   Z←1
[2]  TEST: →(TOLERANCE≥|X-Z*2)/0
[3]   Z←Z-((Z*2)-X) ÷ 2×Z
[4]   →TEST
    ∇
```

Later on, students might return to this example to examine it in more detail and attempt to generalize it. Why, they might ask, does one take one-half of the relative difference (i.e. why divide by $2\times Z$)? Is the point that efficient convergence is obtained by "splitting the difference"?

If when this question arises students have been introduced to slopes and derivatives, they are ready to recognize that the divisor 2 is not an arbitrary constant, but is the derivative of the square function. The function *SQRT* can be treated as a special case of something more general by replacing the constant 2 by an explicit indication of the derivative, and applying it to any polynomial. That will give us a much more general function *ROOT*, based upon exactly the same principles employed in the square route routine from which we started. (Since a polynomial may have several roots, the function takes a left argument to indicate a starting point.)

```
    ∇ Z←START ROOT C
[1]   Z←START
[2]  TEST: →(TOLERANCE≥|(C POLY Z)-0)/0
[3]   Z←Z-((C POLY Z)-0)÷(DERIV C) POLY Z
[4]   →TEST
    ∇

    ∇ Z←C POLY X
[1]   Z←(X∘.*¯1+⍳⍴C)+.×C
    ∇

    ∇ Z←DERIV C
[1]   Z←(⍳⍴C) × (1↓C),0
    ∇
```

The phrase `(C POLY Z)-0` is included to make obvious the parallel between the program to converge on a zero of the polynomial and the expression `(Z*2)-X` in converging on a square root of `X`; obviously `(C POLY Z)-0` can be replaced by `C POLY Z`. The divisor `2×Z` in `SQRT` is equivalent to `(DERIV C) POLY Z`. A new definition of `SQRT` could be written in terms of `ROOT`:

```
    ∇ Z←SQRT X
[1]   Z←1 ROOT (-X),0 1
    ∇
```

since the square root of `X` is the zero of the polymonial with coeffcients `(-X), 0, 1`. The square root of 3 can be found as a zero of the polynomial x^2-3, represented in APL as the polynomial with coefficients `¯3 0 1`:

```
      1 ROOT ¯3 0 1                SQRT 3
1.732050808                1.732050808
```

7. Independent Variation of a Function's Arguments.

Outer products provide more general use of a function, by permitting it to consider all possible pairings (i.e. the Cartesian product) of its arguments.

If a function has two or more arguments, it will often be possible to write its definition so that the two arguments can be independently varied. For example, a business class studying compound interest might state the growth of a unit investment at rate `R` over time `T` thus:

```
    ∇ Z←R GROWTH T
[1]   Z←(1+R)*T
    ∇
```

Then the value of 24 dollars invested at 6% for 300 years would be

```
      24 × .06 GROWTH 300
9.375E8
```

Using that definition, students could study the effect of varying *T* for a particular *R*, or the effect of varying *R* for a particular *T*, but it would be difficult to vary both together. However, if its definition is modified slightly, the function can evaluate all combinations of an array of rates and an array of times:

```
    ∇ Z←R GROWTH T
[1]   Z←(1+R)∘.*T
    ∇
```

Then a table of growth coefficients for (say) each of the rates from 1% to 10% for each of the time intervals from 1 to 50 is generated by

```
      (.01×⍳10) GROWTH ⍳50
```

Exhaustive calculation of all possible combinations can also be obtained with outer products. For example, if *N* students take a test with *Q* questions, their answers might be entered into a *N*-by-*Q* matrix called *X*. If the answers were multiple-choice from among a set *A*, then the incidence array considering all possible combinations of students, questions, and answers is given by `I←X∘.=A`. The sum indicating the number of students who gave each possible answer to each question would be `+⌿I`; the cross-tabulation of answers by question would be `(⍉I)+.∧I`.

To test by exhaustion a proposition about the behavior of a function over some finite domain, students might employ an outer product. Suppose, for instance, they are inquiring whether the function minus is associative for a set of possible arguments *S*. To pick out a particular trio of members of *S* we might create *A*, *B*, and *C* as follows:

```
A←S[?⍴S]
B←S[?⍴S]
C←S[?⍴S]
```

If minus is an associative function, for any three members of *S* it will always be true that `(A-(B-C)) = ((A-B)-C)` .

To test all possible permutations of the three elements of *S* which might be selected, we have only to rewrite that last expression as an outer product:

$$(S\circ.-(S\circ.-S)) = ((S\circ.-S)\circ.-S)$$

Preceding that expression by ∧/, tests whether it is true throughout, and hence whether the function is associative for *S*.

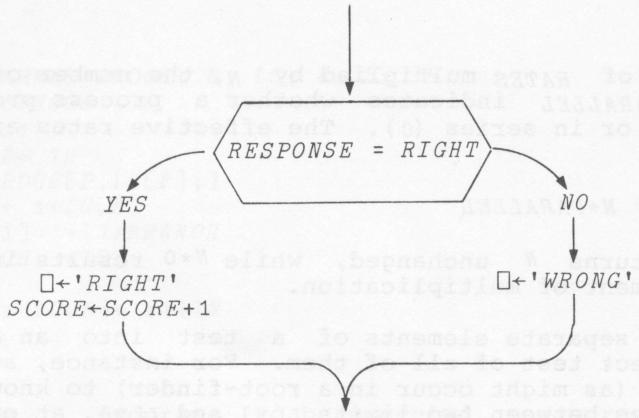
8. Keep branch structures relevant. Needless branching distracts from the structure of a program. It may be avoided by effective use of the identity elements of functions, and by referring to arrays rather than to their separate elements.

Some processes are inherently iterative, and are necessarily written with a test to determine whether another iteration should be made, followed by branches to alternate parts of the program depending on the outcome. Some algorithms require the program to identify which of several quite different procedures applies to a particular case, and to select the appropriate portion of the definition. These are properly represented by branching programs, and their structure is succinctly described by flow diagrams.

There are many other algorithms which could be written either with or without branching. The structure of a definition is more readily apparent when branching is reserved for choices fundamental to the procedure, and otherwise avoided.

Branches are made unnecessary by selecting a single statement which covers appropriately all the different cases. This usually means writing an expression in terms of an entire array, so that all members of the array are considered at once. In the simplest case, a function is to be applied to some members of an array but not others. Stating it another way, the function is to be applied to an array in such a way that some members of the argument appear unchanged in the result. This is done by encoding the test that indicates whether or not the function is required so that the indication "not required for this member" is an identity element of the function.

An interactive drill program, for example, may pose a problem to a student and receive an answer. If the answer is right, he should receive a message confirming that fact, and 1 should be added to his score. If the answer is wrong, he should receive a different message, and 1 should not be added to his score. This could be represented as a branch:



The same effect could be achieved without branching by thinking of the sequence as "Send appropriate message and add appropriate amount to score":

```
OK←RESPONSE=RIGHT
MESSAGE[1+OK;]
SCORE←SCORE+OK
```

The logical comparison of *RESPONSE* with *RIGHT* is added to the score, exploiting the fact that the result 0 for false is the identity element of addition. The 0 or 1 value resulting from the comparison is also used to index the two-row matrix of messages.

A program for the calculation of payroll might provide that each employee who works more than 40 hours receives time-and-a-half for the excess. This could be treated as a branch: if hours do not exceed 40, pay is *RATE*×*HOURS*; if hours exceed 40, pay is *RATE* × *HOURS* + .5×*HOURS*-40. The two expression can be combined without branching:

```
PAY← RATE × (40⌊HOURS) + (.5×0⌈HOURS-40)
```

Moreover, this single expression remains valid when applied to an array of hours and rates, whereas a branching statement would have to be applied to each case individually.

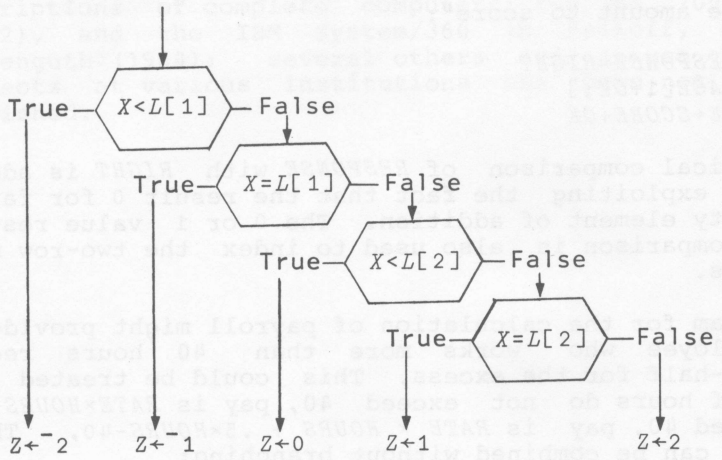
Different functions have different identity elements. For example, the identity element for multiplication is 1; the requisite 1s may be produced by exponentiation. In a queuing model, the rate at which various processes are completed might be represented by a vector *RATES*. Some of these processes are serial, but others are parallel. For a process that works serially, the effective rate is simply the appropriate member of *RATES*. But for a parallel process, the effective rate is

the element of *RATES* multiplied by *N*, the number of cases. The vector *PARALLEL* indicates whether a process proceeds in parallel (1) or in series (0). The effective rates are found by

```
RATES × N*PARALLEL
```

since *N**1 returns *N* unchanged, while *N**0 results in 1, the identity element of multiplication.

Combining separate elements of a test into an array may permit a direct test of all of them. For instance, suppose it is necessary (as might occur in a root-finder) to know whether a value *X* lies between two limits *L*[1] and *L*[2], at one or the other of them, or beyond them. A succession of branches could select one of five possible outcomes:



The same result is produced by

```
Z←+/×X-L
```

Moreover, that expression is not dependent on the order of *L*[1] and *L*[2], whereas the branching structure just shown requires that the lower limit be in *L*[1]. The branch-free representation could be made to apply to an array of *X* values and, if need be, to an array of limits by:

```
+/×X∘.-L
```

9. Distinguish the Function from the Vehicle. Keep the key functions separate from input-output and other housekeeping.

This is an application of the principle that a topic should be divided into relevant functions which take arguments and return results. Since formatting of output or conversation with the user are often desirable, provision for these features should be included in some outer function, distinct from the central algorithm. By way of illustration, here is the definition of a function which plays the game "Life," a cellular automaton devised by J H Conway, reported by Gardner (1970). The "life form" lives in a boundless Cartesian plane, some of whose cells are occupied by "life" while the others are empty. This may conveniently be represented as a logical matrix, with 1s for the occupied cells and 0s for the others. At each generation, new individuals are born into certain empty cells, while some of the previously occupied cells are vacated by death. The object is to examine those configurations which die, or expand, oscillate, or have other collective fates:

```
    ∇ LIFEGAME; I; X
[1]   X←ENTER
[2]   I←0
[3]  LOOP: →(~∨/,X)/END
[4]   X PRINT I←I+1
[5]   X←LIFE BORDERED X
[6]   →LOOP
[7]  END: 'EXTINCTION'
    ∇
```

ENTER is a conversational function to accept the initial image. *PRINT* displays the pattern at generation *I*, converting the logical matrix *X* to a double-width array of characters in which the original 1s and 0s are replaced by circles and blanks. The function *BORDERED* assures that the borders of the pattern are empty by adding an additional empty row or column at the edges if needed.

The heart of the program is in statement 5, which uses the function *LIFE*. *LIFE* contains the rules for the growth of this automaton, while *LIFEGAME* provides the conversational framework in which the game is played. The definition of *LIFE* follows Conway's English-language statement of the rules:

```
    ∇ Z←LIFE X; N; BORN; DIED
[1]   N←NEIGHBORS X
[2]   BORN←N=3
[3]   DIED←X∧(N>3)∨(N<2)
[4]   Z←BORN∨X∧~DIED
    ∇
```

This requires only the function *NEIGHBORS*, which calculates the number of occupied cells adjacent to each cell.

Separate Definitions for the Important and the Trivial

In the examples in the preceding pages, we have advocated creating separate definitions for sub-functions that in more traditional programming would probably not have been treated as distinct subroutines. In particular, we have advocated the definition of a separate function, with formal arguments and result, for each of the important or useful concepts of the topic being studied. We have argued that this gives the student an operational statement of the various functions that he needs to understand, and does so in a way that is both readable and executable, and hence can be the subject both of analysis and of empirical investigation.

In a few cases, we have also illustrated another situation which we believe calls for the definition of separate functions. That is where some commonplace activity is required, not crucial to the topic at hand, but which in order to maintain generality would require an APL expression that might distract the reader from the central point. For instance, in several applications one wishes to join two arrays which may or may not have matching dimensions. It is a simple matter to have the smaller expanded to the size of the larger, but one which, if expressed solely as in APL primitives, will require a phrase as long as all the rest of the definition. We have preferred to put such details into auxiliary functions such as *ON* and *AND*, used here and in the following sections to form a matrix by joining two arrays:

```
    ∇ Z←A ON B;M
[1]   A←MATRIX A
[2]   B←MATRIX B
[3]   M←0,¯1↑(⍴A)⌈(⍴B)
[4]   Z←((M⌈⍴A)↑A),[1] (M⌈⍴B)↑B
    ∇

    ∇ Z←MATRIX A
[1]   Z←((×/⍴A) ⌊ (×/¯1↓⍴A), ×/(¯1+⍴⍴A)↓⍴A) ⍴A
    ∇

    ∇ Z←A BESIDE B
[1]   Z←⍉(⍉A) ON ⍉B
    ∇
```

In summary, we have used definitions whenever an expression seemed either important enough or trivial enough to warrant it.

2. USES OF FUNCTIONS IN A PHYSICAL SCIENCE COURSE

In the foregoing, we have considered how important concepts in a discipline may be treated as functions, and can be given APL definitions that are at once formal, readable, and executable. Applying this approach to an experimental science raises some new issues, which we will consider next. To illustrate these in a specific example, we consider the design of a unit on kinematics as it might appear in an elementary course on physics.

Suppose we wish students to understand the motion of falling bodies, as a first step in the study of acceleration. Position and time are related by a familiar formula `P←.5×G×T*2` in which *T* is the length of time during which the acceleration (in this case, free fall) continues, *G* is the acceleration due to gravity, and *P* is the resulting position. This function, or its inverse `T←(P÷.5×G)*.5` in which elapsed time is stated as a function of position, appears in every elementary course.

The task of a course on physics is not simply to supply students with these idealized formulas, but also to give them some inkling of how scientists go about discovering such things. To embody idealized formulas in APL functions would be easy enough, and might indeed offer some opportunities to explore their implications. But going directly to ideal formulas both conceals the complexity of the real situation and neglects the process of research by which such abstractions are teased from nature.

The scientist starts with the assumption that natural phenomena can be described and understood by "natural laws," and that those natural laws are, in some sense, "simple" functions. His job is to deduce, from an examination of the results of his experiments, what those functions are. In this sense, nature is the author of functions whose definitions are unknown, but whose behavior may be observed and whose rules may be inferred. However, the behavior of these functions is never directly observable. Observations are always mediated by the experimental apparatus and the tools of observation. In mathematics, a student may supply some arguments to a function, find the corresponding results, and ponder what they imply. But at the very simplest, the scientist is faced with a situation in which his observations are related to his independent variable by some composition of functions, such as:

```
OBS ← ERRORFN APPARATUSFN LAWFN X
```

In effect, the researcher's job is to find what statement they can make about the definition of a physical law after separating out the effects introduced by the apparatus and the errors of measurement.

A Concrete Example: Dropping Weights

To introduce these procedures to students, one might start with a simple situation in which they can directly carry out observations themselves. At the same time, a topic should be selected that will permit students a vivid impression of the physical situation that is to be described by a formal model. As an introduction to a unit on the general laws of motion, students might study the acceleration of gravity by timing the fall of weights dropped from various heights.

For the range of heights available inside a room, the times of fall are so small that they are difficult to measure. A more feasible range may be found by dropping weights from the windows of a building having several floors. For a first experiment, the following procedure is proposed: One experimenter stands at ground level while a fellow experimenter goes in turn to a window on each of the various floors. The observer at the ground calls a starting signal to his assistant at the window, and simultaneously sets in motion a stop-watch. When the weight strikes the ground, he stops the watch. He records the time of fall for each of several heights.

As a first step in analyzing these observations, students may tabulate observed fall-time versus height. They are invited to apply techniques already familiar from algebra for deducing the definition of a function from a table of its properties. For instance, they may construct a mapping showing how points in the domain (the various heights) are linked to the corresponding points in the range (fall times). Some functions are readily distinguished by the patterns that appear when such a mapping is constructed (see Iverson 1972, chapter 10).

But when a mapping is constructed for these observations, a problem appears which was not present in the mappings of mathematical functions: the mapping lines appear jumbled and confused. Typical observations made for three falls from each of the heights 2.5, 5, 7.5, and 10 meters are mapped as shown in Fig. 1.

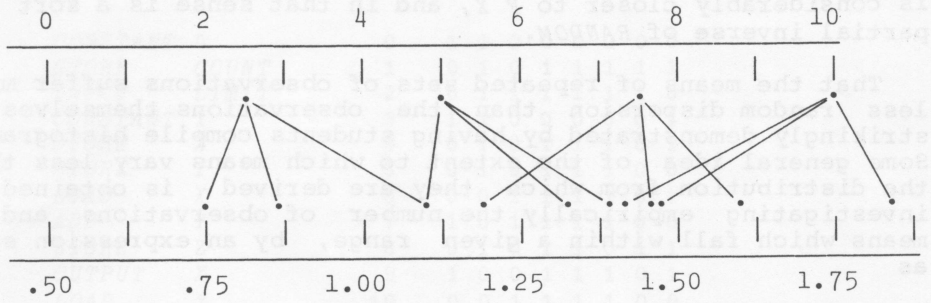


Fig. 1. Mapping from height in meters to time in seconds, for twelve observations of falling weights.

As students repeat observations, so that that there are multiple observations from a single height, they find that a single point in the domain is connected to several different points in the range. This is not at all consistent with the behavior of a lawful function; indeed, the fact that a value in the domain can lead to various values of the range is evidence that this is not a function in the usual sense.

Random Error and its "Inverse"

The problem, of course, is that the observations are subject to error. In an experiment such as this one the errors are large compared to the systematic effects of acceleration by gravity, and mask whatever pattern might otherwise emerge. Before students can go any further in attempts to construct models (i.e. to define functions) that describe the motion of falling bodies, they must deal with some problems of statistics.

Students wishing to observe *F X* instead observe *RANDOM F X*, in which the function *RANDOM* adds or subtracts some random amount from the true value of *F X*. If we could say how much was added or subtracted in each instance, we would be able to state an inverse to the function *RANDOM*, and *RANDINV RANDOM F X* would be equivalent to *F X*. Clearly a genuine inverse random function is impossible. But if *X* consists of repetitions of the same value, and if we are willing to treat any one element as substitutable for any other,

```
(ρX)ρ MEAN RANDOM F X
```

is considerably closer to *F X*, and in that sense is a sort of partial inverse of *RANDOM*.

That the means of repeated sets of observations suffer much less random dispersion than the observations themselves is strikingly demonstrated by having students compile histograms. Some general idea of the extent to which means vary less than the distribution from which they are derived is obtained by investigating empirically the number of observations and of means which fall within a given range, by an expression such as

```
+/RANGE ≥ |TIME - MEAN TIME
```

or, if several values of *RANGE* are employed,

```
+/RANGE ∘.≥ |TIME - MEAN TIME
```

The dispersions of the individual observations may then be compared with those of data obtained when each element of *TIME* is itself a mean of several observations. The relative dispersion of averaged data and of single-observation data can then be compared by computing for each the percentage of members of *TIME* falling within a given interval around *MEAN TIME*. This sort of investigation, requiring large numbers of repetitions, is tedious to carry out, but the needed data are readily generated by a simulator supplied by the instructor.

Requirements for a Simulator

Students may use the computer to analyze the observations, and they will be expected to use the computer to execute the models that they construct to summarize and explain their observations. Now we introduce the computer in yet another role: as the source of data that students will analyze. Our general philosophy that programs should directly reflect the concepts of the discipline implies criteria for the simulator's design:

1. The students should be able to control how the simulator is used to generate observations.

2. The action of the simulator should correctly mirror the relevant aspects of the experiment, and in particular should force students to come to terms both with observational error and with the effects of the procedure by which observations are made.

3. The inner structure of the simulator (when revealed) should correctly and recognizably state the principles that students are to discover.

These requirements are met using a general-purpose outer simulator, together with specialized functions representing the relevant laws and the procedure used to observe them. The law functions and the apparatus functions are defined so that they cannot usefully be executed alone. To simulate an experiment, students must enter a phrase such as

```
SIMULATE 'FALL' MANUALTIMED HEIGHTS
```

This expression yields an array of results having the same shape as *HEIGHTS*, each element representing an observation with that apparatus and the indicated height. Executing any of these functions separately produces no useful result; for instance, *MANUALTIMED* simply returns as its result whatever argument it is given. But as side effects it establishes the parameters of law and apparatus which will be interpreted by *SIMULATE*, whose definition is both general and simple:

```
    ∇ Z←SIMULATE X
[1]   Z← DETERMINISTIC + ERROR
    ∇
```

The two components of *SIMULATE* are:

```
    ∇ Z← DETERMINISTIC
[1]   Z← (⍎LAW) + ⍎SYSTEMATIC
    ∇
```

```
    ∇ Z← ERROR
[1]   Z← ⍎RANDOM
    ∇
```

The symbol ⍎ means that a literal expression is now to be executed. The variables *LAW*, *SYSTEMATIC*, and *RANDOM* are given values as side effects of the apparatus function *MANUALTIMED*; they contain references to the appropriate functions. A possible definition for *MANUALTIMED* might be

```
    ∇ Z←LAWFN MANUALTIMED X
[1]   LAW←'0 0 0 ',LAWFN,' X'
[2]   SYSTEMATIC←'0.2 + INVSOUND X'
[3]   RANDOM←'0 0.15 DISTR X'
[4]   MEASURE←'TIME'
[5]   MEDIUM←AIR
[6]   Z←X
    ∇
```

This definition (presumably locked during students's initial use of it) shows that the initial time, position, and velocity are to be considered 0. The procedure is subject to

systematic effects because of the use of a sound signal, and a systematic delay due to the response time of the human observers. Thus *SYSTEMATIC* includes a constant .2 seconds plus a component inversely related to the speed of sound. Observations are affected by random errors with mean 0 and standard deviation .15 secs. The measure of interest is time, and the medium through which the fall takes place is air.

When the simulator is used with an appropriate apparatus function, the variable *LAW* will contain and expression such as '0 0 0 *FALL X*'; the definition of *FALL* (see below) makes use of very general statements of the laws of motion.

The family of definitions for use with *SIMULATE* is an example of a style of programming more devious than those we illustrated in the last chapter. The function *MANUALTIMED* does not itself explain how the manually operated timer converts the input heights to resulting observed times, but serves instead to evoke unseen definitions which the instructor supplied earlier. Thus, while students will eventually be able to see and understand these inner definitions, the outer layer to which they have immediate access deliberately conceals some of the underlying detail. We do this in order to give students a sequence of experiences that will permit them to discover some key principles for themselves, and build for themselves definitions to represent their discoveries.

Analysis of a Source of Systematic Error

To examine effects introduced by the apparatus or procedure, students must be encouraged to consider variations in them, and to search for systematic effects that they may introduce. Suppose that in order to improve the accuracy of recording, an electrical device is arranged so that iron weights suspended by an electromagnet are dropped by the flick of a switch which both releases a weight and starts an electric timer. This procedure is represented by a function *ELECTRICTIMED*. It will become apparent that the two procedures yield results which differ both in variance and in value. That is, the change in apparatus has not only reduced some sources of error, but has made some systematic difference as well. Exploring the nature of this systematic difference offers a number of rewarding possibilities.

In order to study the effect of varying the height from which weights are dropped with repeated observations at each height, students may make *X* a matrix. For instance:

```
X←⍳100 10⍴HEIGHT←3×⍳10
ME←MEAN SIMULATE 'FALLTIME' ELECTRICTIMED X
MM←MEAN SIMULATE 'FALLTIME' MANUALTIMED X
```

Then they may compare the set of means *ME* based upon the electrical timer with the set *MM* based upon the manual timer. These may be tabulated by an expression such as

```
ME AND MM AND HEIGHT
```

or their difference plotted as a function of height by the expression:

```
PLOT (ME - MM) VS HEIGHT
```

If students pursue this analysis far enough, it will ultimately become apparent that the difference of the means obtained by the two procedures is linearly related to height. They may calculate the slope of this line, and the height of the point at which it crosses the vertical axis. Are these simply exercises in analytic geometry? No, for students may discover a physical meaning for each. The slope of the difference between the results of the two procedures is (roughly) the reciprocal of the speed of sound. The constant term by which the manually-obtained measures lag behind the electrical ones is an estimate of the lag introduced by the reaction-times of the experimenters. In this way students may be able not merely to control the effects of certain sources of systematic error, but in doing so to measure other phenomena of general interest.

Student-Constructed Models

Students should be encouraged to try various models, which may either state falltime as a function of height or distance fallen as a function of time. Depending upon what the unknown underlying function is, a polynomial may be appropriate in one sense but not in the inverse, or may be appropriate to neither. The next figure shows the means of the preceding observations, superimposed upon plots of two proposed functions.

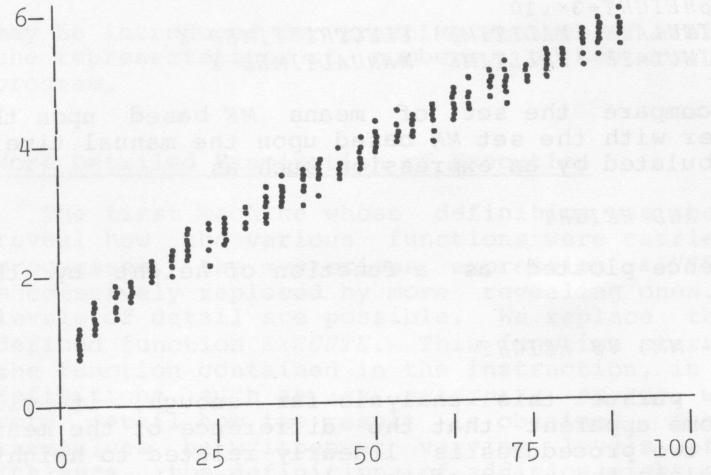


Fig. 2. Simulated falltimes, time in seconds vs. height in meters for 100 observations at 3-meter intervals from 3 to 90 meters.

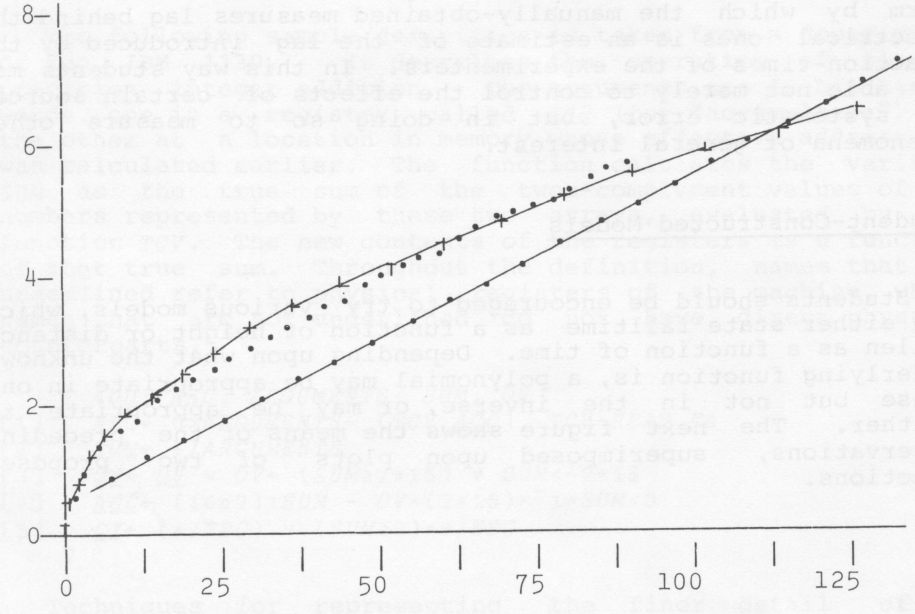


Fig. 3. Observations and two hypothetical curves showing time of fall in seconds vs. height in meters. The crosses show height as the function of time `H←3×T*2`, while the straight line of dots shows time as the function of height `T←0.5 + 0.05×H`. Each of these is shown extrapolated slightly beyond the range of the observations.

Detailed simulation of a falling body is a matter of considerable complexity (involving the mass and air resistance of the weight, as well as the error-prone procedure for measuring the time), so it is unlikely that students can with these initial models find close agreement either with their own observations or with the instructor's simulator. Students should nevertheless understand that the models they construct are attempted summaries of the phenomenon; the more sophisticated simulations differ in detail but not in underlying approach to the functions they propose. The consideration of alternative models proposed by students may, if the instructor wishes, lead to reflections on procedures for measuring goodness of fit, or even to discussion of the nature of explanation in the philosophy of science.

Considering the Relation in the Opposite Sense

The initial exercises involved attempts to state the time of fall as a function of the height from which the weights were dropped. Students may also consider the inverse function, that is, position as a function of fall-time. The order in which the problem was originally stated (time as a function of height) was arbitrary. No matter which way the relationship is considered, it will ultimately be necessary to consider its inverse in order to solve equations (for instance, to determine from what height a weight must be dropped in order to experience a given fall-time). Ultimately, the instructor will want to lead students to a general treatment of motion as the integral of the forces acting on a body over time.

To develop the investigation of position as a function of time, it will be useful to change the apparatus to record the position the weight has reached at the end of various intervals of time. The experimenters affix marks showing the distance downward from some moderately high window to the ground. A camera which takes high-speed photographs at rapid intervals is mounted so that the entire distance of the fall is within its view. A series of photographs taken in this way shows the weight at various moments during its fall, against a background from which it is possible to read its position. To use this device, it is necessary to establish the time interval at which the camera will take pictures, and to start the camera when the weight is released.

The design of the camera (or the statement of functions that represent its contribution to the experiment) brings to the fore a question that was ignored in the first experiments. Errors of measurement occur not only in the recording of the

positions of the weights in the photographs, but also in determining the instant at which the camera takes each picture. The independent variable, as well as the recording of the dependent variable, is subject to error. In an elementary presentation, this new source of error may be ignored. Alternatively, the manner in which a change in *X* produces a change in *F X* may be the subject of new investigations, opening many possibilities for considering the slope of a function, or more generally the derivatives of simple functions.

Introductory Experiences with Curve-Fitting

From experiments (or simulations) with the camera, students will eventually obtain data pairing each average position with a corresponding set of average intervals. They may be invited to venture some models of their own, in an effort to produce a summary of the important aspects of the relation they have observed. After they have constructed some models by guesswork, they may be led to consider more systematic procedures for fitting certain functions to their observations.

Iverson (1972, Chapters 10 and 14) mentions that the coefficients of a factorial polynomial can be obtained by inspection of a function's difference table. Suppose a function such as *D* finds the difference between successive members of a vector:

```
    ∇ Z←D X
[1]   Z←(1↓X) - ¯1↓X
    ∇
```

To construct a difference table for a function *F*, students evaluate *F X* when *X* is the consecutive integers starting with 0. In the successive columns of the difference table they place *F X*, then *D F X*, then *D D F X*, and so on. Then the top row of the difference table divided by !0 1 2 3... is the set of coefficients for a factorial polynomial. If the difference function *D* is re-applied until no further difference remains --i.e. as many times as there are observations-- coefficients will be found for a polynomial whose degree is one less than the number of observations, and which fits the observed data precisely.

Finding polynomial coefficients from a difference table is subject to the difficulty that the method makes no allowance for error. The *X**0 term is based on a single observation, the *X**1 term on the difference between a single pair, and so on.

Where many observations are collected, the additional data have no effect upon the low-order terms of the polynomial. In particular, if a polynomial of low degree would suffice as a model of the data but the observations are subject to error, the additional observations do nothing to improve the faulty estimates of the low-order coefficients. Students may therefore be introduced to various schemes for pooling the estimates from different portions of a difference table. The way is open to introduce students to a variety of schemes for assessing the goodness of fit of a fitted polynomial, and for refining the estimates of their coefficients.

Having indicated in a general way the nature of the problem, the instructor may lead students to the use of the APL primitive ⌹, which gives the least squares solution for a set of over-determined linear equations. If this is applied to the successive powers of *X*, (as it was in definition of *FIT* in the preceding chapter) the result is the set of coefficients of the best-fitting polynomial. By applying a function such as *FIT* to their observations, students may obtain the coefficients of a polynomial of any degree up to one less than the number of observations. The coefficients of a polynomial of degree 5 for position as a function of time would be obtained from an expression such as:

```
      5 FIT TIME AND POSITION
```

Applying the Fitted Curve to New Data

If the camera being used to record positions (or the function that simulates it) divides the time of fall into ten intervals and a mean position is found for each, it becomes a simple matter to find coefficients for best-fitting polynomials up to the ninth degree. This may be an embarassment of riches! How may students decide which if any of these is reasonable?

As a first approximation, they may plot the observed data and the data found from evaluating the polynomial versus the ten time units. Of course, the polynomial of the highest degree will reproduce the observed data perfectly; all the polynomials other than those of degree zero (the mean) and degree one (a straight line) will fit reasonably well.

However, if any of the polyomials found by this procedure is an approximation to a more general law, that polynomial will serve not only to fit the particular observations on which its coefficients were based, but also to fit new data. So the next task is to obtain new data, for fall-times beyond

the range used to find the polynomial's coefficients. The task is then to compare the new data with points derived from evaluating the former polynomial. With this procedure, it will become apparent that polynomials of high degree give grossly inappropriate answers for values of *X* outside the range on which their coefficients were based. Only the second degree polynomial continues to give an adequate approximation. By this procedure, students may be led to the impression that a second degree polynomial is a rather good bet for the position function, and warrants further investigation.

In the pages that follow, plots are shown for polynomials of degree 1, 2, 3, 4, and 5, for two sets of observations of a falling weight. The positions are recorded at intervals of 0.2 seconds. Two sets of observations are made, *P1* and *P2*. *P1* contains the observations for times .2 through 2.0 sec. Polynomials of various degrees were fitted to express position as a function of time. Then a further 10 observations *P2* were obtained, corresponding to times 2.2 through 4.0 sec. Finally, the fitted polynomials were evaluated for *T*, all twenty times employed, i.e. .2 through 4.0 secs. The fitted curve thus represents an extrapolation to 4 seconds of the best-fitting polynomial based upon times up to 2 seconds. The graphs on the following five pages were generated by the function *COMPARE*:

```
    ∇ COMPARE DEGREE; COEFF
[1]   COEFF←DEGREE FIT (10↑T) AND P1
[2]   PLOT ((P1,P2) AND T POLY COEFF) VS T
    ∇
```

The lines linking the crosses that indicate the fitted polynomial were drawn in afterwards by hand.

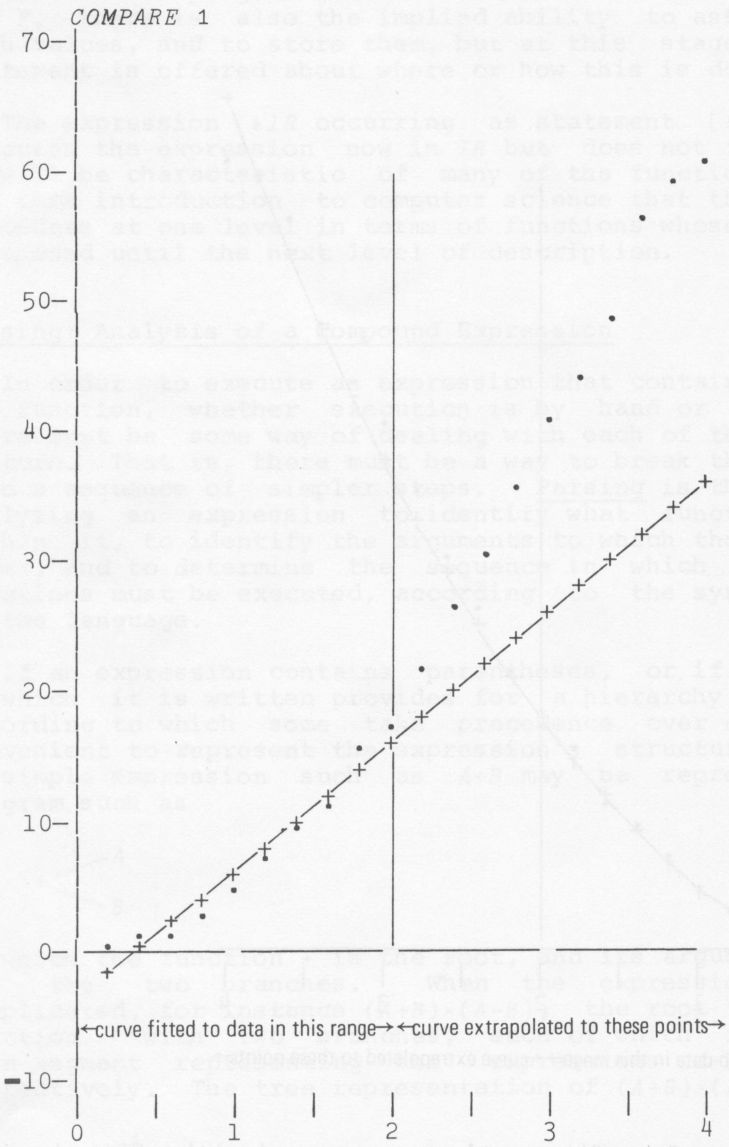


Fig. 4. Observations compared with fitted and extrapolated curves, showing distance in meters as a function of time in seconds, for a polynomial of degree 1.

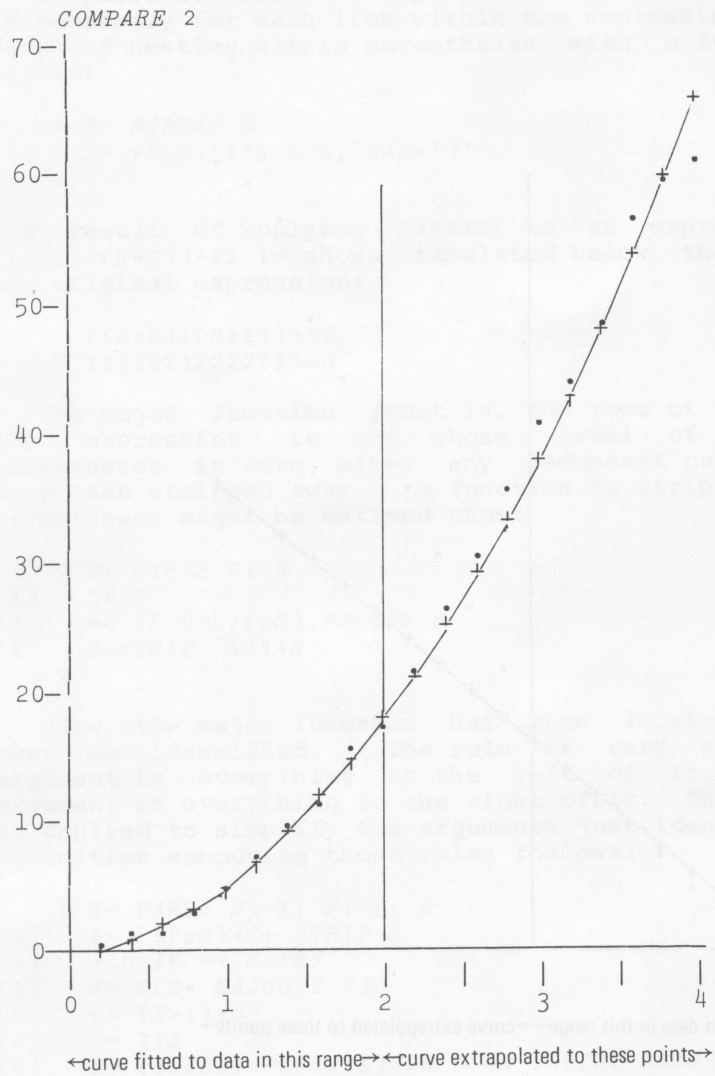


Fig. 5. Observations compared with fitted and extrapolated curves, showing distance in meters as a function of time in seconds, for a polynomial of degree 2.

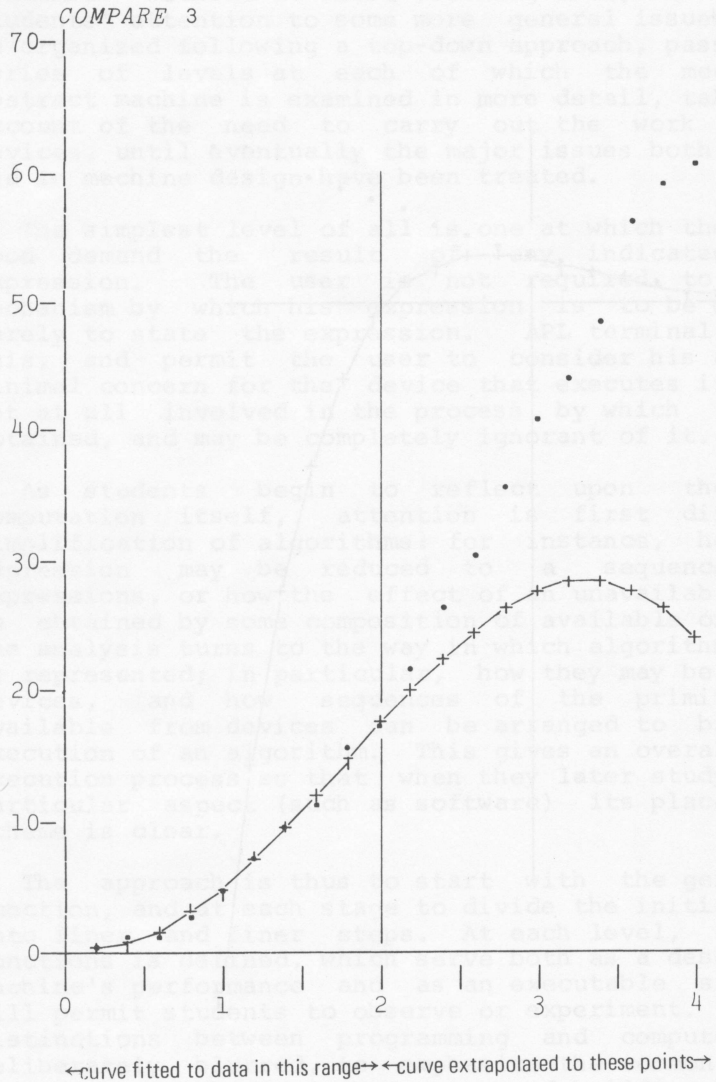


Fig. 6. Observations compared with fitted and extrapolated curves, showing distance in meters as a function of time in seconds, for a polynomial of degree 3.

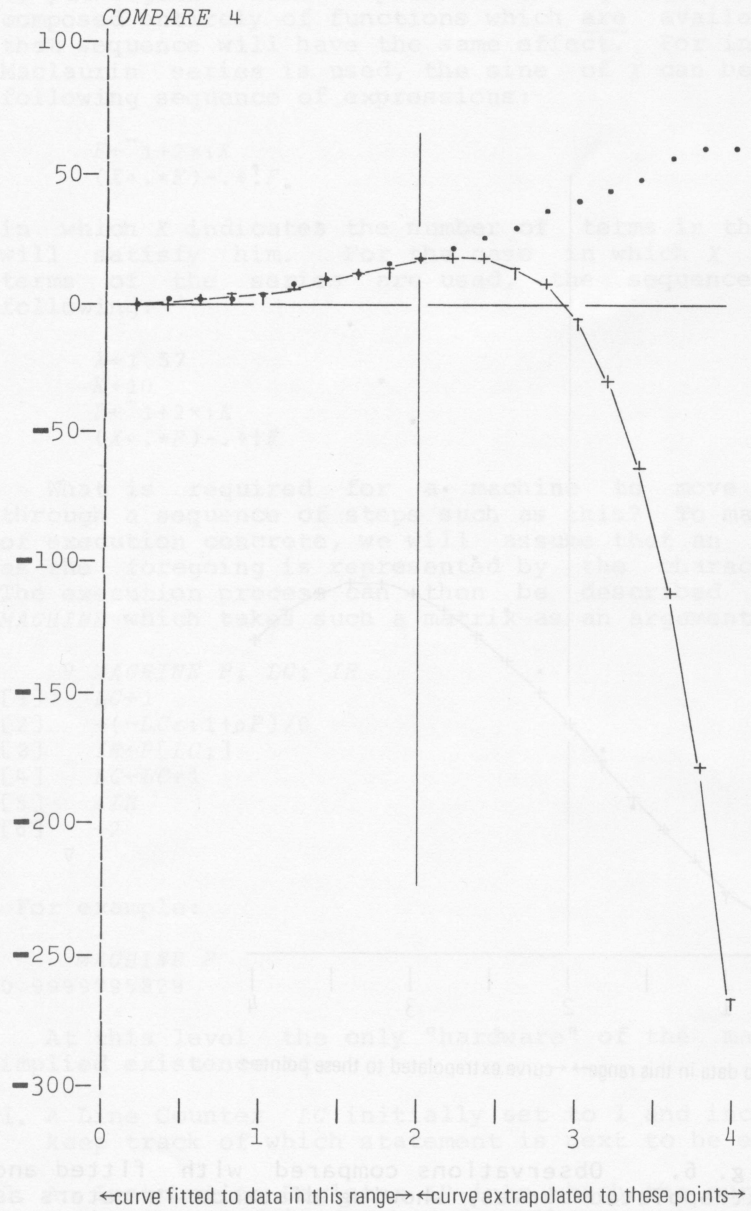


Fig. 7. Observations compared with fitted and extrapolated curves, showing distance in meters as a function of time in seconds, for a polynomial of degree 4.

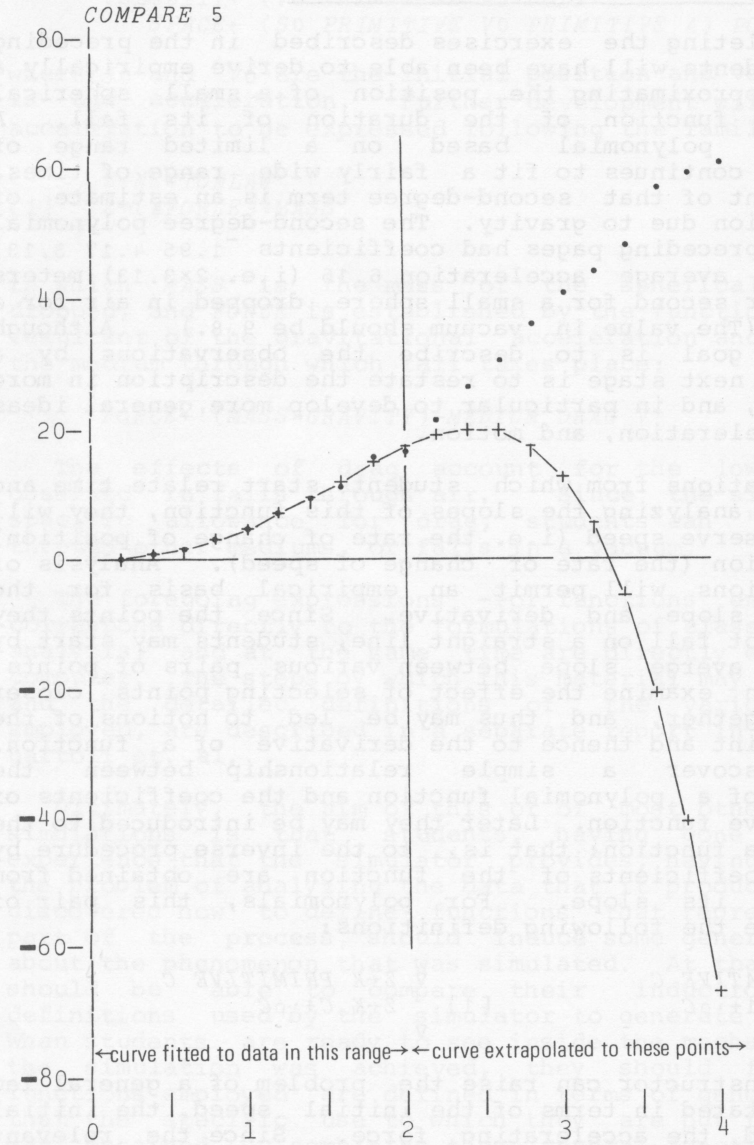


Fig. 8. Observations compared with fitted and extrapolated curves, showing distance in meters as a function of time in seconds, for a polynomial of degree 5.

Further Analysis of the Functions of Motion

After completing the exercises described in the preceding sections, students will have been able to derive empirically a polynomial approximating the position of a small spherical weight as a function of the duration of its fall. A second-degree polynomial based on a limited range of observations continues to fit a fairly wide range of times. The coefficient of that second-degree term is an estimate of the acceleration due to gravity. The second-degree polynomial shown in the preceding pages had coefficients ¯1.95 4.17 3.13, indicating an average acceleration 6.16 (i.e. 2×3.13) meters per second per second for a small sphere dropped in air for a short time. (The value in vacuum should be 9.8.) Although the initial goal is to describe the observations by a function, the next stage is to restate the description in more general terms, and in particular to develop more general ideas of force, acceleration, and motion.

The observations from which students start relate time and position. By analyzing the slopes of this function, they will be able to observe speed (i.e. the rate of change of position) and acceleration (the rate of change of speed). Analysis of the observations will permit an empirical basis for the concepts of slope and derivative. Since the points they observed do not fall on a straight line, students may start by finding the averge slope between various pairs of points. Then they can examine the effect of selecting points closer and closer together, and thus may be led to notions of the slope at a point and thence to the derivative of a function. They may discover a simple relationship between the coefficients of a polynomial function and the coefficients of its derivative function. Later they may be introduced to the primitive of a function: that is, to the inverse procedure by which the coefficients of the function are obtained from knowledge of its slope. For polynomials, this pair of functions have the following definitions:

```
    ∇ Z←DERIVATIVE C
[1]   Z←1↓C×¯1+⍳⍴C
    ∇
```

```
    ∇ Z←K PRIMITIVE C
[1]   Z←K,C÷⍳⍴C
    ∇
```

Now the instructor can raise the problem of a general law of motion, stated in terms of the initial speed, the initial position, and the accelerating force. Since the relevant information about velocity and acceleration provides the derivatives of a function of motion, the appropriate function is obtained by using the function *PRIMITIVE* to calculate the coefficients of a polynomial, and then evaluating it for the desired values of *T*. In that way, the student is able to reach the general expressions

```
      VELOCITY← (V0 PRIMITIVE A) POLY T
         SPACE← (S0 PRIMITIVE V0 PRIMITIVE A) POLY T
```

where *S0* and *V0* are the initial position and velocity, and *A* is the acceleration. Further development will then permit acceleration to be expressed following the familiar formula:

```
      Z←NEWTONLAW
[1]   Z←(±FORCE)÷MASS
   ∇
```

in which *MASS* is the mass of the spherical weight being dropped, and *FORCE* is established by the function *FALL* as the resultant of the gravitational acceleration and the drag of the medium through which fall takes place:

```
      FORCE←'(MASS×GRAVITY)-MEDIUM DRAG V'
```

The effects of drag account for the low acceleration observed in falls through air. Since the simulator makes specific allowance for drag, students can "observe" falls through other mediums, or falls in a vacuum.

The foregoing expressions --or functions embodying them-- correspond directly to the formulations of classical kinematic analysis, and at the same time are directly executable at a computer. The steps by which this material may be developed, and the detailed definitions of the various functions employed, are described in a separate report in preparation by Bartoli et. al.

Ultimately, the aim of this or of most other educational simulations is that students, having gone through the experience that the simulator provides, having grappled with the problem of analyzing the data that it produces, and having discovered how to define functions that represent at least part of the process, should induce some general principles about the phenomenon that was simulated. At that point, they should be able to compare their inductions with the definitions used by the simulator to generate data for them. When students are ready to see inside the mechanism by which the simulation was achieved, they should find that the functions employed are defined in terms of general laws, and that the specific use to which they are put in the exercise was obtained by combining these general functions with more specific ones (such as *MANUALTIMED*) which serve to supply appropriate values for the various parameters. In this way, they find that the functions used for the simulation are directly transferable to a wide range of new applications, and that the definitions employed within them are consistent with the more general statement of the lesson.

3. AN ABSTRACT MACHINE FOR THE INTRODUCTION TO COMPUTER SCIENCE

An important idea stressed in the foregoing pages is that the language used to state concepts and to execute them at a terminal -- APL -- is independent of the mechanism used to evaluate expressions written in it. This independence frees the user from having to consider the mechanism by which the computer carries out its instructions, and from having to concern himself with machine-imposed characteristics that are irrelevant to the topic he is studying. But suppose he wants to study computer science? It is possible to develop in APL the definitions of functions which correspond to the principal functions of the device to be studied. A course can be developed for students who have some general familiarity with writing and executing expressions written in APL, but as yet to have given little thought to formalizing the rules by which he does so, or to the organization of devices for executing them automatically.

Algorithms for the Execution of Algorithms

Doing what a user wants -- (i.e. evaluating a user-written algorithm) is a central purpose of any computer system. The basic concepts of computing may be introduced by considering what performance is required (be it of a person or of a machine) when an algorithm is executed.

Programming is the art of building the definitions of new functions from a set of previously defined functions. Computer science extends that domain to include the special problems of building those definitions when the set of available functions rests ultimately in the physical representation of information and the function of host systems, and ultimately to the characteristics of physical devices, i.e., the computer hardware.

Successive Levels of the "Abstract Machine"

Our analysis starts from a quite general level with what we have called an abstract machine. The abstract machine is a set of functions for carrying out calculations in a way that corresponds to the basic organization of a computing machine but which (at first) has no specified physical form. It responds appropriately to requests without revealing its own mechanism. The description of this abstract machine makes frequent use of global concepts such as "evaluate" or

"remember" without at first defining them, in order to direct students' attention to some more general issues. Topics can be organized following a top-down approach, passing through a series of levels at each of which the mechanism of the abstract machine is examined in more detail, taking increasing account of the need to carry out the work with physical devices, until eventually the major issues both in programming and in machine design have been treated.

The simplest level of all is one at which the user receives upon demand the result of any indicated function or expression. The user is not required to describe the mechanism by which his expression is to be evaluated, but merely to state the expression. APL terminal systems permit this, and permit the user to consider his algorithm with minimal concern for the device that executes it. The user is not at all involved in the process by which the results are obtained, and may be completely ignorant of it.

As students begin to reflect upon the process of computation itself, attention is first directed to the simplification of algorithms: for instance, how a compound expression may be reduced to a sequence of simpler expressions, or how the effect of an unavailable function can be obtained by some composition of available ones. Gradually the analysis turns to the way in which algorithms and data may be represented; in particular, how they may be represented in devices, and how sequences of the primitive functions available from devices can be arranged to bring about the execution of an algorithm. This gives an overall view of the execution process so that when they later study in detail a particular aspect (such as software) its place in the larger scheme is clear.

The approach is thus to start with the general notion of function, and at each stage to divide the initial descriptions into finer and finer steps. At each level, a family of APL functions is defined, which serve both as a description of the machine's performance and as an executable simulation which will permit students to observe or experiment. At this level, distinctions between programming and computer science are deliberately blurred, to emphasize the common basis of both branches of the study of ways of building and executing functions.

Program: Obtaining the Effect of an Unavailable Function

Suppose that students need to use a sine function, but that means to evaluate that function are not provided. The task is

to put together some expression or sequence of expressions, composed entirely of functions which are available, so that that sequence will have the same effect. For instance, if the Maclaurin series is used, the sine of X can be found by the following sequence of expressions:

```
      F←¯1+2×⍳K
      (X∘.*F)-.÷!F
```

in which K indicates the number of terms in the series which will satisfy him. For the case in which X is 1.57 and 10 terms of the series are used, the sequence becomes the following:

```
      X←1.57
      K←10
      F←¯1+2×⍳K
      (X∘.*F)-.÷!F
```

What is required for a machine to move automatically through a sequence of steps such as this? To make the problem of execution concrete, we will assume that an algorithm such as the foregoing is represented by the character matrix P. The execution process can then be described by a function *MACHINE* which takes such a matrix as an argument:

```
    ∇ MACHINE P; LC; IR
[1]   LC←1
[2]   →(~LC∈⍳1↑⍴P)/0
[3]   IR←P[LC;]
[4]   LC←LC+1
[5]   ⍎IR
[6]   →2
    ∇
```

For example:

```
    MACHINE P
0.9999995829
```

At this level the only "hardware" of the machine is the implied existence of

1. A Line Counter *LC* initially set to 1 and increased by 1 to keep track of which statement is next to be executed.

2. An Instruction Register *IR* into which the current statement to be executed is fetched from the memory.

3. A Program Memory, here represented by the argument P, containing the list of statements to be executed.

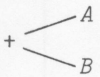
Since the program sets values for variables called *X*, *K*, and *F*, there is also the implied ability to associate names with values, and to store them, but at this stage no explicit statement is offered about where or how this is done.

The expression ⍎*IR* occurring as statement [5] of *MACHINE* executes the expression now in *IR* but does not indicate how. It will be characteristic of many of the functions developed for this introduction to computer science that they explain a procedure at one level in terms of functions whose details are postponed until the next level of description.

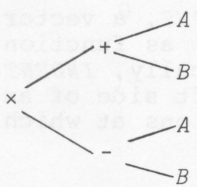
Parsing: Analysis of a Compound Expression

In order to execute an expression that contains more than one function, whether execution is by hand or by machine, there must be some way of dealing with each of the functions in turn. That is, there must be a way to break the expression into a sequence of simpler steps. Parsing is the process of analyzing an expression to identify what functions appear within it, to identify the arguments to which those functions refer, and to determine the sequence in which the various functions must be executed, according to the syntactic rules of the language.

If an expression contains parentheses, or if the language in which it is written provides for a hierarchy of functions according to which some take precedence over others, it is convenient to represent the expression's structure as a tree. A simple expression such as *A+B* may be represented by a diagram such as



in which the function + is the root, and its arguments *A* and *B* are the two branches. When the expression is more complicated, for instance *(A+B)×(A-B)*, the root is the major function × with two branches, each of which is itself a tree-segment representing the expressions *A+B* and *A-B* respectively. The tree representation of *(A+B)×(A-B)* is:



To parse correctly an expression containing parentheses, it is necessary for each item within the expression to find the depth of nesting within parentheses with a function such as *PARENS*:

```
    ∇ Z←PARENS S
[1]   Z←+\(S='(')-0,¯1↓S=')'
    ∇
```

The result of applying *PARENS* to an expression such as ((A*2)+(B*2))*÷2 is shown tabulated below the characters of the original expression:

```
      ((A*2)+(B*2))*÷2
      1222221222221000
```

The major function (that is, the root of the tree) of an APL expression is one whose level of nesting within parentheses is zero, after any redundant outer parentheses have been stripped away. A function to strip redundant outer parentheses might be defined thus:

```
    ∇ Z←STRIP S;N
[1]   Z←S
[2]   →0 IF 0=⌊/(⍴S),PARENS S
[3]   Z←STRIP ¯1↓1↓S
    ∇
```

Once the major function has been located, its arguments must be identified. The rule is very simple: its left argument is everything to the left of it, and its right argument is everything to the right of it. Then the same rule is applied to simplify the arguments just identified. An APL definition embodying those rules follows:

```
    ∇ Z←PARSE S;X;F;L;R
[1]   Z←(1⌈⍴S)↑S←STRIP S
[2]   →0 IF ~∨/S∈FNS
[3]   F←S[X←MAJORFN S]
[4]   L←(X-1)↑S
[5]   R←X↓S
[6]   Z←(INDENT PARSE L) ON F ON INDENT PARSE R
    ∇
```

This definition makes use of the global variable *FNS*, a vector of those characters which are to be recognized as function symbols. The function *ON* joins two arrays vertically, *INDENT* inserts an arbitrary number of spaces at the left side of an array, and *INDEX* returns the indices of all positions at which a proposition is true.

When an expression contains two or more functions whose level within parentheses is zero, which one of them is the major function? Conventions differ. In conventional algebra and in languages such as FORTRAN, there is a hierarchy of precedence. A function of high precedence acts as though it and the variables immediately adjacent to it were surrounded by unseen parentheses. Thus in those languages the function with lowest precedence is the major function. APL has no precedence hierarchy, and so in APL the major function is determined solely by position.

When precedence rules are insufficient to determine which is the major function (because several functions have the same precedence, or because the language, like APL, has no precedence), the final decision is made on the basis of position. Conventional algebra is ambiguous on this point; in FORTRAN, the major function is the *LAST* eligible function, while in APL it is the *FIRST*. These rules of order and precedence are evident in the definition of *MAJORFN*, which returns the index of the major function in an expression:

```
    ∇ Z←MAJORFN S
[1]   LEVEL←(PREC⍳S)+(1+⍴PREC)×PARENS S
[2]   Z←ORDER↑INDEX(S∈FNS)∧LEVEL=⌊/LEVEL
    ∇
```

The variable *PREC* contains a list of the function symbols, in order of precedence from lowest to highest. For an APL expression, *PREC* is an empty vector. The variable *ORDER* has the value 1 when the first function is to be selected (as in APL), and ¯1 when the last is to be selected (as in FORTRAN).

In the result produced by *PARSE*, the arguments of a diadic function appear indented one level to the right of their function, with the left argument above the function and the right argument below. For a monadic function, the line immediately above it (i.e. where its left argument would otherwise be) is left blank. Two examples of the use of *PARSE* follow; the lines linking the various nodes were drawn by hand. The first shows parsing of an APL expression for the roots of a quadratic:

```
      ORDER←1
      PREC←''
      STRING←'(-B+1 ¯1×((B*2)-4×A×C)*÷2)÷2×A'
```

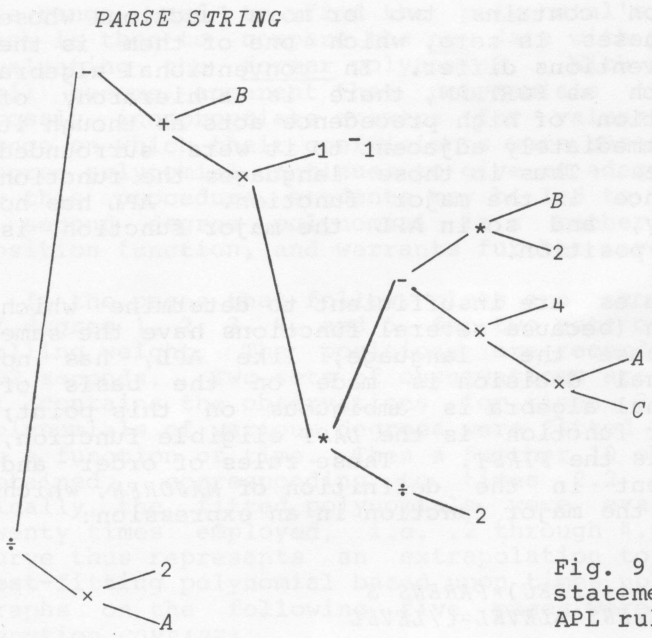


Fig. 9. Tree diagram showing statement parsed by APL rules.

Parsing of a FORTRAN statement may be illustrated with an expression involving polynomials, whose conventional representation exploits the precedence rules:

```
ORDER←¯1
PREC←'-+÷××'
PARSE 'A×X*3+B×X*2+C×X+K'
```

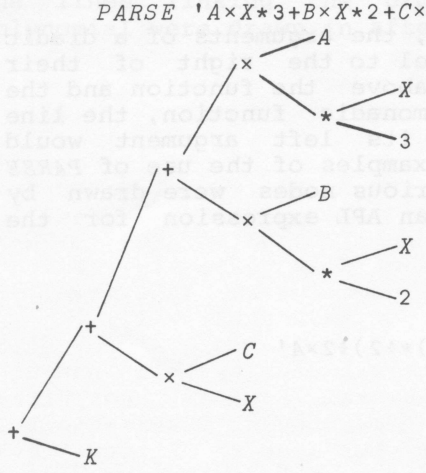


Fig. 10. Tree diagram showing statement parsed by FORTRAN rules.

Compilers and Machines for Executing Compiled Programs

A compiler takes a statement (or program) written in one language and produces from it a sequence of statements more acceptable to the machine that will execute them. The tree diagrams generated by *PARSE* could be used to compile a program. Working from the leaves toward the root of the tree, each node forms an expression consisting of a single function and its arguments. Each of these may be given a name to identify its result in temporary storage. An expression such as

```
STRING←'(-B+1 ¯1×((B*2)-4×A×C)*÷2)÷2×A'
```

may thus be transformed to a tree by a function such as *PARSE*, and thence by a function *TREECOMPILE* (definition not shown) to a set of simple expressions arranged in the same tree structure. By collecting those statements in order from the leaves toward the root, a compiled program is obtained, executable by a function such as *MACHINE*:

```
      PROGRAM←TREECOMPILE PARSE STRING
      PROGRAM
A←A×C
B←B*2
A←4×A
A←B-A
B← ÷2
A←A*B
A←1 ¯1×A
A←B+A
A← -A
B←2×A
□←A÷B
```

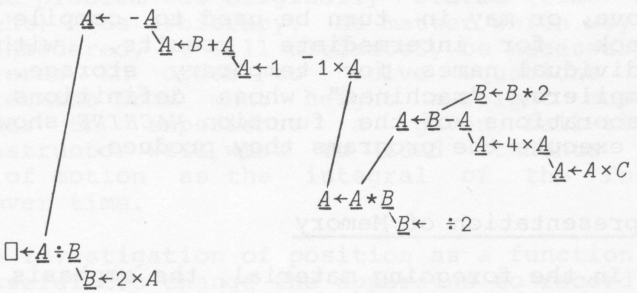


Fig 11. Names assigned to nodes of a tree and collected to form a compiled program.

Parenthesis-Free Notation

A tree serves to indicate graphically the syntactic structure of an expression. The same information can be conveyed in linear form in the Lukasiewicz parenthesis-free notation, sometimes called Polish notation. In Polish form, both arguments of a function follow the function, so that $A+B$ is written $+AB$, and $(A+B)\times C$ is written $\times+ABC$. An expression in Polish form is readily executed by following these rules:

1. Locate the last function in the string. Its arguments appear immediately to the right of it.

2. Replace the function and its arguments by their value. Repeat until all functions have been evaluated.

A function *POLISH*, which converts an APL statement (or a FORTRAN or similar statement) to Polish form, can be given a definition exactly parallel to the definition of *PARSE*. The first five statements are unchanged; only the order in which the components are assembled is different:

```
    ∇ Z←POLISH S;X;F;L;R
[1]   Z←(1⌈⍴S)↑S←STRIP S
[2]   →0 IF ~∨/S∈FNS
[3]   F←S[X←MAJORFN S]
[4]   L←(X-1)↑S
[5]   R←X↓S
[6]   Z←F,(POLISH L),POLISH R
    ∇
```

The resulting parenthesis-free string may be executed by another simple machine built to follow the two rules stated above, or may in turn be used to compile a program using a stack for intermediate results, without the need for individual names for temporary storage. For each of these compilers, "machines" whose definitions are variations or elaborations of the function *MACHINE* shown above can be used to execute the programs they produce.

Representation of Memory

In the foregoing material, the emphasis was on analyzing a function into a sequence of statements, and then on analyzing each statement into several simpler ones, until each step consisted of a single function and its arguments. The memory of the abstract machine was implied by the ability to assign values to names and later to refer to those variables by name, but no explanation of the procedure was offered. As the analysis moves closer to the physical embodiment of the machine, students can examine in more detail a function for evaluating simple expressions. It assumes that an expression presented to it consists of five elements, such as *A←B+C*, in which the third and fifth elements are names for the arguments, the fourth elements is the name of the function to be executed, and the first is the name of the destination to which the result will be assigned.

```
    ∇ RESULT← EVALUATE S; LEFT; RIGHT; RESULT
[1]   LEFT← FETCH S[3]
[2]   RIGHT← FETCH S[5]
[3]   RESULT← EXECUTE S[4]
    ∇
```

This definition identifies the next two areas for study:

1. The functions which substitute the appropriate values for the names of the arguments; that is, which fetch an item from memory, and which store into memory the result when it is found.

2. The functions which, once the values of the arguments are at hand, can execute the appropriate function and calculates a result.

The result produced by *EVALUATE* is usually passed directly to a function which stores the result in the memory:

```
    S[1] STORE EVALUATE S
```

The previously unexplained references to variables by their names must now be replaced by a memory access program. Further specification of the way this is done will emerge from the definitions of functions such as *FETCH* and *STORE*. Memory may be treated as an array, having a physical extent, so that items in it may be identified by their positions in the array as well as by name. A symbol table or dictionary may be used to make the translation between name and location. In the foregoing examples, names were all single characters, but with the aid of a dictionary or symbol table, tokens of uniform size may be used to stand for names of any length.

Representation of Data

It is possible to do considerable work on the organization of memory while still treating the individual items within memory as numbers or letters, without yet specifying how these are represented. However, before the execution of functions can be examined in detail, the manner in which their operands are represented first must be developed. The way in which the encoding of numbers influences the evaluation of functions is an extensive and important field of numerical analysis and computer science. At this introductory stage, memory may be represented not as a list of names or numbers, but as a logical matrix corresponding to the digits used for the internal encoding of information in many machines. Functions

may be introduced for treating vectors of arbitrary symbols as the representations of numbers, of letters, or of the stored program.

More Detailed Examination of Execution

The first machine whose definition was shown above did not reveal how the various functions were carried out. As study progresses, the mysterious expression *⍎INSTRUCTION* may be successively replaced by more revealing ones. But still many levels of detail are possible. We replace the symbol ⍎ by a defined function *EXECUTE*. This function starts by identifying the function contained in the instruction, in order to call on definitions such as *ADD*, *MULTIPLY*, *DIVIDE*, which specify in more detail how the result is obtained. These functions may themselves be written at varying levels of detail. For instance, the definition of addition might still employ the primitive + but with specific provision to identify the range of the arguments that can be considered in a finite machine, and the setting of overflow or carry indicators. At the next level the definition may go on to specify the sequence of logical functions which produce the effect of addition.

The following sample definition is taken from a description of the IBM 1130. It describes the operation of single-precision integer addition. The arguments are 16-bit binary words, one in a register called *ACC* (for "accumulator") and the other at a location in memory whose effective address *EA* was calculated earlier. The function calulates the variable *SUM* as the true sum of the twos-complement values of the numbers represented by these two arrays, evaluated by the function *TCV*. The new contents of the registers is a function of that true sum. Throughout the definition, names that are underlined refer to physical registers of the machine, while names that are not underlined do not have direct physical counterparts.

```
    ∇ ADD; NEG; ARGUMENTS; SUM; OV
[1]   NEG← 0>ARGUMENTS← (TCV ACC), TCV READ EA
[2]   SUM← +/ARGUMENTS
[3]   OV← OV ∨ OV← (SUM≥2*15) ∨ SUM<-2*15
[4]   ACC← (16⍴2)⊤SUM - OV×(2*16)×¯1*SUM<0
[5]   CY← (∧/NEG) ∨ (SUM≥0)∧≠/NEG
    ∇
```

Techniques for representing the finer detail of an operation such as addition may be suggested by considering the following three definitions for a parallel adder. The first is written for a base-10 device such as might be found in an

adding machine or an odometer. The second is written in exactly parallel fashion, but substituting base-2 for base-10. The third shows another base-2 device, but one in which the arithmetic functions used in the first and second definitions have been replaced by logical functions, to indicate how the effect of the arithmetic function of addition can be obtained from the proper sequence of logical functions:

```
    ∇ Z←A ADD10 B;C
[1]   Z←10|A+B
[2]   C←10≤A+B
[3]   →(C∧.=0)/0
[4]   A←Z
[5]   B←1↓C,0
[6]   →1
    ∇
```

```
    ∇ Z←A ADD2 B;C
[1]   Z←2|A+B
[2]   C←2≤A+B
[3]   →(C∧.=0)/0
[4]   A←Z
[5]   B←1↓C,0
[6]   →1
    ∇
```

```
    ∇ Z←A ADDL B;C
[1]   Z←A≠B
[2]   C←A∧B
[3]   →(C∧.=0)/0
[4]   A←Z
[5]   B←1↓C,0
[6]   →1
    ∇
```

Circuit Logic

Pursuing the description of the mechanism of execution even further, the functions ∧ and ≠, which are critical to the logical adder just defined, may themselves be defined in terms of the logical NAND, a physical component of many computers. Hence for statements [1] and [2] of *ADDL* we could substitute

```
[1]   Z←A NEQ B
[2]   C←A AND B
```

in which the functions *AND* and *NEQ* have the following definitions:

```
    ∇ Z←A AND B
[1]   Z←(A⍲B)⍲(A⍲B)
    ∇
```

```
    ∇ Z←A NEQ B
[1]   Z←(A⍲B⍲B)⍲(B⍲A⍲A)
    ∇
```

The representation of computer circuits and architecture by APL functions has been extensively developed by Hellerman in his text Digital Computer System Principles, first published in 1967. Use of APL functions to represent the elements of electronic circuits is the subject of a text by Spence (1973) and of a set of APL programs developed by Penfield (1971).

Student-Built Models

At each of the levels of the course, students should build functional models. The mechanism of a simple computer can be defined as a family of functions, its output observed, or its work traced at the terminal. To illustrate how one such model

might be constructed, we present a description of a very small and simple machine. It consists of a memory of 32 8-bit binary words, an 8-bit accumulator, an 8-bit instruction register, and a 5-bit instruction counter. The memory is used to store both programs and data. In an instruction, the first three bits are an operation code, and the remaining five are an address. The eight possible operations and their codes are as follows

```
000  Halt
001  Load accumulator from address
010  Store from accumulator to address
011  Input data to address
100  Output data from address
101  Add to acumulator value at address
110  Constant from instruction to accumulator
111  Branch to address if accumulator unequal to contents
     of last memory location.
```

The machine is represented by a single function, whose definition follows:

```
    ∇ MACHINE; EA; UNEQ
[1]   IC← 0 0 0 0 0
[2]  NEXT:   IR←M[1+2⊥IC;]
[3]   IC←(5ρ2)⊤1+2⊥IC
[4]   EA←1+2⊥3↓IR
[5]   →HALT+2⊥3↑IR
[6]  HALT:   →0
[7]  LOAD:   →NEXT, A←M[EA;]
[8]  STORE:  →NEXT, M[EA;]←A
[9]  INPUT:  →NEXT, M[EA;]←(8ρ2)⊤⎕
[10] OUTPUT: →NEXT, ⎕←2⊥M[EA;]
[11] ADD:    →NEXT, A←(8ρ2)⊤(2⊥A)+2⊥M[EA;]
[12] CONST:  →NEXT, A← 0 0 0 ,3↓IR
[13] BRANCH: →NEXT, IC←(IC∧~UNEQ)∨3↓IR∧UNEQ←A∨.≠M[1+31;]
    ∇
```

(The machine here described uses 0-origin indexing to refer to locations in its own memory. However, the APL description is written in 1-origin.)

A program must be represented in the machine by a set of binary words. A program to print the first *N* terms of the Fibonacci series is shown below. At the left is a listing showing the names of the operations involved and designating names for the variables used during the calculation. At the right are shown the corresponding binary codes by which the program is represented in the machine:

```
CONSTANT 0          0   1 1 0 0 0 0 0 0
STORE    COUNT      1   0 1 0 1 1 1 1 1
INPUT    LIMIT      2   0 1 1 1 1 1 1 0
CONSTANT 1          3   1 1 0 0 0 0 0 1
STORE    X          4   0 1 0 1 1 1 0 1
STORE    Y          5   0 1 0 1 1 1 0 0
LOAD     X          6   0 0 1 1 1 1 0 1
ADD      Y          7   1 0 1 1 1 1 0 0
STORE    Z          8   0 1 0 1 1 0 1 1
OUTPUT   X          9   1 0 0 1 1 1 0 1
LOAD     Y         10   0 0 1 1 1 1 0 0
STORE    X         11   0 1 0 1 1 1 0 1
LOAD     Z         12   0 0 1 1 1 0 1 1
STORE    Y         13   0 1 0 1 1 1 0 0
CONSTANT 1         14   1 1 0 0 0 0 0 1
ADD      COUNT     15   1 0 1 1 1 1 1 1
STORE    COUNT     16   0 1 0 1 1 1 1 1
LOAD     LIMIT     17   0 0 1 1 1 1 1 0
BRANCH   6         18   1 1 1 0 0 1 1 0
HALT               19   0 0 0 0 0 0 0 0
```

To show this machine in action, the program shown at the right is placed in the first twenty rows of the memory matrix *M*, and started by typing *MACHINE*. The symbol □: indicates the request for input, in this case the number of terms of the Fibonacci series to be printed:

```
      MACHINE
□:
      6
1
1
2
3
5
8
```

It is a considerable chore to prepare a program in the binary form in which it is used by the machine. The student may ease his task by constructing a simple assembler. Then each instruction can be written in a more convenient form. The following assembler will accept a symbolic program such as the one shown above. It recognizes the eight opcodes by the first letter in the name of each. Constants and branch instructions must have a numeric operand, but memory addresses (i.e. variable names) may be of any length.

```
    ∇ Z←ASSEMBLE P
[1]   Z← 0 8 ρI←0
[2]   SYMBOLS← 0 0 ρ''
[3]  TEST: →((1↑ρP)<I←I+1)/0
[4]   OP←1↑STATEMENT←P[I;]
[5]   ARG←(STATEMENT⍳' ')↓STATEMENT
[6]   Z←Z ON (OPCODE OP), OP ENCODE ARG
[7]   →TEST
    ∇
```

The function *OPCODE* translates the mnemonic operation codes to binary:

```
    ∇ Z←OPCODE OP
[1]   Z←, 2 2 2 ⊤8|¯1+'HLSIOACB'⍳OP
    ∇
```

The argument must also be translated. If the opcode is *B* or *C*, or if the argument consists solely of numerals, the translation is direct. But if the argument is a variable name, it is translated by the function *LOCATION* which builds and consults a symbol table.

```
    ∇ Z←OP ENCODE ARG
[1]   ARG←(ARG≠' ')/ARG
[2]   →((OP∈'BC')∨∧/ARG∈'0123456789')/NUMERIC
[3]   Z←,(5ρ2)⊤LOCATION ARG
[4]   →0
[5]  NUMERIC: Z←,(5ρ2)⊤10⊥¯1+'0123456789'⍳ARG
    ∇

    ∇ Z←LOCATION ARG
[1]   SYMBOLS←SYMBOLS WITH ARG
[2]   Z←32-(SYMBOLS∧.=(¯1↑ρSYMBOLS)↑ARG)/⍳1↑ρSYMBOLS
    ∇
```

Before reporting where a particular name is found in the symbol table, the function *WITH* is used to make sure that the name is included:

```
    ∇ Z←SYMBS WITH ARG; OLD
[1]   Z←SYMBS ON ARG
[2]   ARG←,(¯1,¯1↑ρZ)↑Z
[3]   OLD←∨/(¯1 0 ↓Z)∧.=ARG
[4]   Z←(-OLD,0)↓Z
    ∇
```

The assembler assigns address for variables starting at the last position in memory and working back. In this example, the symbols and their numeric addresses were:

```
   31 COUNT
   30 LIMIT
   29 X
   28 Y
   27 Z
```

In order to trace the execution of this program, a function *TRACE* could be inserted between statements 4 and 5 of *MACHINE*:

```
    ∇ Z←TRACE
[1]   Z←2 DIGIT ¯1+2⊥IC
[2]   Z←Z,' ','HLSIOACB'[1+2⊥3↑IR],2 DIGIT 2⊥3↓IR
[3]   Z←Z,' ',3 DIGIT 2⊥M[EA;]
[4]   Z←Z,' ',3 DIGIT 2⊥A
    ∇

    ∇ Z←N DIGIT X
[1]   Z←'0123456789'[1+(Nρ10)⊤X]
    ∇
```

TRACE displays before execution of each instruction the value of the program counter, the instruction (as a symbolic op code), the value of the operand, and the value of the accumulator. A tracing of the Fibonacci program is shown below; the input and output appear embedded in sequence.

```
00 C00 192 004      06 L29 001 004      13 S28 003 005
01 S31 004 000      07 A28 002 001      14 C01 095 005
02 I30 004 000      08 S27 002 003      15 A31 002 001
□:                  09 O29 001 003      16 S31 002 003
      4             1                   17 L30 004 003
03 C01 095 000      10 L28 002 003      18 B06 061 004
04 S29 005 001      11 S29 001 002      06 L29 003 004
05 S28 008 001      12 L27 003 002      07 A28 005 003
06 L29 001 001      13 S28 002 003      08 S27 005 008
07 A28 001 001      14 C01 095 003      09 O29 003 008
08 S27 008 002      15 A31 001 001      3
09 O29 001 002      16 S31 001 002      10 L28 005 008
1                   17 L30 004 002      11 S29 003 005
10 L28 001 002      18 B06 061 004      12 L27 008 005
11 S29 001 001      06 L29 002 004      13 S28 005 008
12 L27 002 001      07 A28 003 002      14 C01 095 008
13 S28 001 002      08 S27 003 005      15 A31 003 001
14 C01 095 002      09 O29 002 005      16 S31 003 004
15 A31 000 001      2                   17 L30 004 004
16 S31 000 001      10 L28 003 005      18 B06 061 004
17 L30 004 001      11 S29 002 003      19 H00 192 004
18 B06 061 004      12 L27 005 003
```

Representing Concurrent Processes

Many devices, institutions or systems are composed of sub-systems which have separate and more or less independent existences of their own, although at times they interact. The overlapping of central-processor functions with input-ouput functions is common in computers, and so the issue arises when we describe computers. But it is also an issue in the simulation of many other devices or systems. For these reasons it is most useful to have techniques for representing processes that are interacting and concurrent.

The description of devices usually reveals two characteristics that did not appear in the programs heretofore discussed:

1. The sub-systems have a continuous existence, so that it is difficult to describe them solely by the results they produce in response to individual presentations of an argument. Falkoff, Iverson, and Sussenguth (1964) coined the term system program to refer to functions that describe such continuously-operating sub-systems.

2. The separate system programs are able to communicate, so that although they behave independently much of the time, the action of one may alter the behavior of another.

The point of contact or interface between two system programs appears as a set of variables occurring in both. Usually a value for such a variable specified in one program is referenced in the other. To denote such a variable, Falkoff, Iverson and Sussenguth (op. cit.) coined the term shared variable.

The continuous functioning of a device can be depicted in the system program which represents it by making the definition an endless loop in which the shared variable takes on new values in response to external events. The shared variable is in effect the function's argument; instead of being set once at the time the function is called, it is set or used repeatedly during the function's endless execution.

Conversational programs are often written in this way. There the effect of a shared variable is provided by the symbol ⍞ or ⎕, indicating that a new value is to be obtained from the terminal. Endlessly looping functions do not have to run up an infinite bill, since the system that executes them may impose the discipline that whenever a shared variable is referenced execution is suspended unless or until it has received a new value.

A Juke Box as an Example of Concurrent Execution

A juke box is a familiar device which may be thought of as a collection of concurrent but interacting functions. The definitions below show an outline for the action of the player mechanism, and the action of one of the push-button stations from which customers indicate their choices.

```
     ∇ PLAYER
[1]    →(~∨/X)/1
[2]    P← (⍴X)|P+(P⌽X)⍳1
[3]    PLAY P
[4]    X[P]←0
[5]    →1
     ∇
```

```
     ∇ SELECTOR
[1]    →(PAID≤0)/1
[2]    S← BUTTONS⍳1
[3]    X← X∨S∈⍳⍴X
[4]    PAID← PAID-1
[5]    →1
     ∇
```

The first line of each of these definitions is a dwell. That is, some condition which is controlled by an event external to the program is tested, and if no exception is found, the program repeats the test. It dwells on that line indefinitely until an exception is encountered. This was the form in which interlocks were represented by Falkoff, Iverson & Sussenguth (1964). If the exchange of values between the two programs were controlled by automatic interlocking, such as that provided by the APL system for ⎕ or ⍞, the dwelling effect might be obtained without repetition.

The logical array *X* is its memory, indicating which tunes are to be played. The function *PLAYER* tests to see whether any record has been selected; if none (read ~∨/ as "not any"), statement 1 is repeated. The rest of the program is entered only after at least one selection has been made.

The variable *P* is a pointer indicating the number of the last record that was played. Once any selection is detected, *P* is reset as the index of the next member of *X* that is on (i.e. has value 1) following *P*. In some models of juke box, the physical mechanism for doing this provides a circular array of contacts, and a rotating arm which, once a selection is registered, advances in a clockwise direction from wherever it last halted. Statement [2] of *PLAYER* shows this wrap-around stepping to the next member of *X* that is "on"; it is written in 0-origin indexing. Statement [3] of *PLAYER* calls a function *PLAY* whose definition is not shown, but which presumably activates the playing of record *P*. When playing is completed, the *P*th member of *X* is turned off, and control returns to statement [1], where the program checks anew to see if any member of *X* is on.

Meanwhile, the selector has a continuous existence of its own. Its first statement is a dwell until *PAID* shows a credit. Then it sets a variable *S* to indicate which of the array of buttons has been pressed. The corresponding member of *X* is set to 1, and the counter *PAID* is reduced by one.

Simple as this description is, one can deduce from it a number of points about the behavior of this juke box. For instance, we note that if the customer presses two buttons simultaneously after making a payment, only one selection is registered: the one with the lower numeric value.

When the selection is transmitted to the array *X*, it is ORed with the previous value of *X*. This means that if the same selection is made twice (as might easily happen if there are several customers) there is no provision to add the number of requests. Although each customer may be satisfied because his selection will indeed be played, their two payments do not result in two playings. If an especially enthusiastic customer asks to hear again the selection that is currently playing, he will be disappointed, since that position in *X* is reset to zero when playing is completed.

We could also deduce from statement [2] of *PLAYER* that the order in which selections are played depends on the selection played last (that is, the value of *P*) and the fixed sequence in which the records are represented by *X*, but not on the sequence in which the buttons are pressed.

This simple illustration should suggest that a formal description of this sort provides a wealth of information about the details of the behavior of the juke box that is unlikely to be obtained from ordinary manuals of operation.

A "Machine" for Executing Several Programs at Once

If definitions such as *PLAYER* or *SELECTOR* were entered in an APL workspace and executed in the usual way, the start of one would prevent execution of the others. However, concurrent executions can be obtained by a program such as *CONCURRENT*. Its argument is a three-dimensional array of programs. Each plane represents a different program, and within any plane, each row is a statement in the program.

```
    ∇ CONCURRENT PROGS; N; LC; P; S; IR
[1]   N← 1↑⍴PROGS
[2]   LC← N⍴1
[3]  NEXT: P← ?N
[4]   IR← PROGS[P;LC[P];]
[5]   LC[P]← 1+LC[P]
[6]   →(IR[1]='→')/BRANCH
[7]  NOBRANCH: ⍎IR
[8]   →NEXT
[9]  BRANCH: LC[P]← ⍎1↓IR
[10]  →NEXT
    ∇
```

There are *N* programs to be executed concurrently. The variable *LC* is a vector of line counters, with one element for each plane of *PROGS*. At the statement labelled *NEXT*, the variable *P* is used to select which of the programs is to be executed next. (In the example, the selection is made at random.) When a statement is selected for execution, its first character is tested to see whether the statement is a branch instruction. If it is, the instruction (lacking its initial →) is evaluated and used to reset the instruction counter for program *P*. Otherwise the instruction is executed as is. Note that this model provides a single instruction register, used in turn by each of the different programs.

This simple definition does not include provision for labels, local variables, or arguments for the various functions that are the successive planes of the array *PROGS*; clearly the definition could be elaborated. A system of this type could be regarded as an elementary time-sharing monitor.

Perspective

We have touched lightly on a number of topics of interest in an introductory course on computer science. The general approach should now be clear: any aspect of the machine execution of algorithms is presented as an explicit function, acting upon an explicit representation of an algorithm. The parts of a system of functions can be made to fit together to produce more complex behavior. At each stage, one has a working model of the system of functions being studied. The very simple functions developed above can be extended in many directions, not only to treat more elaborate parsers or more complex machines, but also to treat entirely different aspects of the problem of algorithm execution.

The entire approach, at any level, is to treat a machine as a function in the formal sense, and to analyze it and describe it in the same way as the subunits or processes within it. It is possible to treat even the component functions from which the micro-instructions of a computer are constructed, and to combine them to form the components of the more global descriptions.

The function definitions produced during these analyses should serve both as reference descriptions and as executable simulations with which students can experiment. A number of aspects of computing have been treated in this way in published articles. Hellerman (1967) used APL functions for several topics in computer logic; Falkoff (1962) treated associative memories, and Falkoff (1972) published an APL functional description of a piano. There are two published descriptions of complete computers: the IBM 7090 in Iverson (1962), and the IBM System/360 in Falkoff, Iverson, & Sussenguth (1964); several others are in use in classroom projects at various institutions but have not yet been published.

4. STUDENT AND MACHINE: EFFECTS OF OPEN USE OF THE COMPUTER

In the examples of programming in the foregoing, APL has been employed in a deliberate effort to transcend the usual role of programming languages. We have used APL's capability not only to simplify the control of computing machines but to simplify the statement of procedures.

The content of various disciplines can be expressed in such a way that APL expressions can be used in books, on paper, and at the blackboard to achieve briefer, clearer, more general and more effective summary than is possible with conventional algebra or the various ad hoc extensions that each discipine tends to develop. At the same time, students can take the APL expressions developed for exposition and, without further translation or instruction, have them executed at a terminal.

Our aim is not to use programming as an end in itself, but rather to integrate the use of algorithms into the study and exposition of any discipline using formal or mathematical models.

Underlying this conception of a common language for man and machine is a conception of the relation between machine and student. In general, we have assumed that if a function is defined, it is because the student is expected both to use it and to understand it. That means that he should expect to be able to display its definition and understand what he reads there. Each function should therefore be not a black box but a glass box, whose inner mechanism is visible to any who need or care to study it.

On occasion we wish temporarily to withhold from a student what the definition of a function is, so that he may practice his analytic powers by testing its performance. We have considered such exercises with locked functions a sort of paradigm of the sciences: we observe the world and collect data on the results that nature's unknown functions provide. The scientist's faith is that the world can indeed be reduced to comparatively simple functions, and that experiment will lead him to propose definitions for functions whose performance will duplicate what he observes. But even in the case in which knowledge of a function's definition is for the time being withheld, the student has the right and the power ultimately to see and to understand what he sees.

We call this way of using the machine open use of the computer. A great deal can be done with no pre-programmed

definitions at all, i.e. with a "bare" APL machine. Generally speaking, it is the student (or the teacher) who initiates activities, not the machine. The task of the student is to develop his definitions of functions and to acquire experience in exploring their consequences. The computer is thus to him a laboratory device, an arena for testing his ideas by evaluating the results his functions produce.

This style of use has far-reaching consequences for the style in which teaching is conducted. Some of these have already been mentioned, and others will be described in more detail in a moment. Here we list some of them:

1. The work with the computer can be integrated into the presentation of the subject matter itself

   a. with minimum attention to programming, machine, language, or computer science (unless those are desired in their own right);

   b. with no requirement that students spend much time individually at the terminal (although if the facility is available it is usually advantageous to do so);

   c. with no dependence on a computer program for the particular course, other than access to an APL system and guidance from text or teacher on the functions that may be developed.

2. The functional approach, using APL to define the functions, has important benefits quite apart from the use of computers.

   a. There are significant advantages of the notation even if no use of terminals is made.

   b. The role of the terminal in many cases is to provide rapid and decisive testing of proposals developed by students; it thereby serves as a powerful motivating device, but not one that is indispensable to the content of instruction.

   c. Even where the terminal is used for carrying out practical computations, the language may still be used apart from the terminal for developing the underlying concepts and procedures.

3. In the classroom, a terminal has many uses as a laboratory device, so that a group can witness the outcome of expressions collectively or individually proposed.

a. Group use, with occasional individual use outside class hours, permits a relatively large number of students per terminal.

b. Terminal manufacturers have not thus far provided for collective viewing of the output of a terminal, and there are few devices well adapted for display to a group. However, adequate results are obtained by mounting a small TV camera on a conventional terminal, and connecting it to one or more TV monitors in the classroom.

4. APL notation and a functional approach are applicable to a wide variety of courses in mathematics and the sciences.

a. The overhead of getting started with the notation and with terminals can be spread over many disciplines.

b. Although direct use of the notation and of terminals can be started with students of junior high school age or even younger, it remains relevant throughout their subsequent secondary and university education.

c. If one anticipated that students would make wide use of APL upon reaching secondary school, many of their elementary courses from kindergarten onward could lay a foundation for the general conception of function and for elements of APL notation, thus making the subsequent introduction of computing even easier.

The Uses of Function Definitions in Teaching

Defined functions in APL have several roles in instruction. We are giving our greatest attention to the technique of selecting the functions to be defined and writing their definitions so that they correspond to the structure of the subject matter. But there are other uses, and it may be useful to indicate their variety in another list:

1. Instructor-defined functions with open definitions.

a. These will usually be presented in a text book or at the black board.

b. The role of the computer is to illustrate or explore their properties, or to permit their use as elements in the definition of new functions.

c. These definitions are intended first to be read and secondarily to be executed by machines; these are the "glass boxes."

d. The instructor may provide in advance many definitions which are useful to the topic being studied but not themselves at issue. These are "primitive" to the level of explanation required by the student, even though not primitive to APL.

2. Instructor-defined functions with locked definitions.

a. "Black box" functions may be provided by having the teacher enter a definition unobserved, locking it to prevent its display. Presumably students are then challenged to discover its properties by experimenting with it, and then are asked to offer definitions that duplicate its behavior.

b. A "black box" which represents a device or natural process may be used to give the student the experience of attempting to respond to it before he understands its structure. This is the usual form of "simulation games." However, we believe that such games can and should be constructed so that student may subsequently inspect, understand, and alter their governing rules.

d. "Black box" definitions may also be used without a terminal, provided at least one person knows how to execute the functions for any argument that may be proposed. This may make the basis of a stimulating classroom contest, in which one team agrees upon a definition and others attempt to deduce what it is by proposing arguments which the other team must evaluate.

3. Student-written definitions.

a. The student learns a great deal attempting to state a formal definition for a familiar procedure.

b. Attempting to state definitions which describe given data is a key aspect of scientific research. Iverson (1972) pays particular attention to strategies for the analysis of functions based upon tables of their results; the discussion of a physical science unit in Part 2 and the paper by Bartoli et al (1973) develop a use of this strategy in introductory physics.

c. Students may readily construct and operate their own models. This may become a major activity both of collective classroom work and as a core of individual projects.

4. Utility functions.

   a. The instructor may find it useful to make available some functions (e.g. in a public library) before their definitions are explained, provided their general effect is clear.

   b. Some utility functions, such as graph plotting and scaling routines, may be used without any effort to explain their working in detail. These utility functions are often written so as to minimize processor time or space requirements, and may therefore use programming techniques that need not be explained.

5. Functions providing ancillary services to the instructor.

   a. Workspaces may be used to keep and analyze class records.

   b. Workspaces may be used to store tests, or to generate fresh data for individual exercises, either for administration at a terminal or else by printing stencils for class use.

6. CAI Author Language.

   a. We have not discussed tutorial CAI in this paper, but clearly an APL system can be used to generate, administer, and record individualized drill or tutorial sessions. To the teacher, this may be in the form of workspaces perhaps entirely prepared elsewhere.

   b. A set of functions may serve as an author language, so that individuals may write their own tutorial programs. The CAL/APL workspaces (Macauley, 1969), developed at Orange Coast College, or the experimental tutorial system developed at Cornell Medical School by Hagamen et al (1971), are examples of this sort of use.

Experiences in Developing Curriculum for Open Use of APL

When the APL interpretive system was first operating at Yorktown Heights, the T. J. Watson Research Center arranged to make APL time available on an experimental basis to a few secondary schools in the vicinity. We were inclined then to believe that all we had to do was to take an experienced classroom teacher, well versed in the material he wanted to teach, and show him how to program in APL. We thought that that would suffice to prepare him to make good use of a computer. We seriously underestimated the problem.

To make effective use of APL in the classroom required a much more complete re-thinking of the curriculum than we had appreciated. It soon became clear that when APL is used to find answers to the sorts of exercises found in conventional texts, the problems are often reduced to triviality. Even texts specifically aimed at "computer math" or "computer science" seemed to be concerned primarily with special characteristics or problems of the computer.

In those early experiments, our excitement with the potential of APL was tempered by frustration at our initial uncertainty of an effective way to apply it to the curriculum. In the absence of a clear guide for students on the use of APL in the main curriculum, many teachers and perhaps a smaller proportion of students lost interest, or engrossed themselves in various elaborate games or data-processing projects unrelated to the long-run goals of either.

Computing contributes most effectively to the process of instruction when its use prompts a thorough restatement of a broad topic. Adding computer exercises to the existing framework of established courses has much less impact.

Far more than we had realized, the language in which concepts are presented and the implicitly assumed technology of computation shape the content of the curriculum. To make good use of APL it was necessary to back away from the curriculum as it was being taught, and seek a fresh restatement. Part of the restatement involved much more direct use of the concept of function. In principle this could have been done with conventional mathematical notation, or by using some of the more informal representation used in systems analysis. But the simplicity of APL syntax coupled with the richness of the set of APL primitive functions made many topics simpler to express than they would have been either in mathematical notation or in other programming languages.

Our principal technique of curriculum development was to seek powerful restatements of the key functions in each discipline. This realization provided a clue to the widening divergence between the direction in which our work was evolving and that taken by our colleagues in CAI. While they were concerned with improving the process of instruction and clarifying just what performance the curriculum requires, we were intent upon restructuring the content and the way the content is represented. Where education development projects start by examining the behavior they must produce and then asking how to obtain it, we have found the greatest dividend in re-examining fundamental concepts, to explore how they can

be stated, and to discover what can be built with them. When that job is done extensively, the order, scope, and presentation of the topics is often greatly changed. It is our belief that many more fields and topics can be treated in that way.

Summary

The APL language permits restatement of the formal or mathematical basis of many scientific topics. Its use of arrays as wholes simplifies both thought and expression.

For each topic to be taught, the key task is developing a language that expresses its fundamental ideas and operations. APL provides a basic syntax within which these specialized languages (i.e. sets of functions) can be defined.

APL should be understood both as a framework for building the definitions of functions, and as a mechanism for exploring the meaning of functions. Backed by a suitable rethinking of the fundamental structure of a topic, these provide powerful tools of expression and computation, and support that combination of experience and analysis which we call "insight."

Appendix
REVIEW OF EARLIER ROLES FOR THE COMPUTER IN EDUCATION

This report has concerned the role that the concept of function can play in teaching, and ways in which APL can be used both to represent functions and to make their meaning immediate and concrete. That is substantially different from earlier conceptions of the way that computing might contribute to education. In this appendix we review those earlier roles in order to indicate why we have chosen to develop a different approach.

Alternative Conceptions of the Computer's Role

Five general roles for the computer in education have emerged to currency in recent years; we will consider them in turn.

1. Accounting.
2. Problem Solving.
3. Simulation.
4. Teaching.
5. Subject of Study.

1. Accounting. In many ways an educational institution resembles a business. The most obvious uses of computing are those that support its administration in common business functions, such as payroll, inventory, class registration, or grade reporting. These may have little effect upon education other than to perform more cheaply, quickly, or impersonally some common routine functions.

A more specialized task sometimes aided by computing is the scheduling of classes; in some cases, the ability to re-schedule frequently may permit schools to adopt more flexible programs, or tolerate greater diversity or more frequent change in the classes their students elect. Giving administrators the power easily to rearrange schedules may thereby make the schools more effective and more open to experimentation -- or at least so some writers have suggested (for instance, Allen, 1967).

A primary goal of many computer educators has been the individualization of instruction, which will be discussed further in the section on computer teaching. But if a "computer tutor" is not available or not feasible,

individualizion may be fostered in another way. Detailed accounting of each student's progress may permit him to follow at his own pace his own route through an array of knowledge or skills. By analyzing reports on each student, the computer may be able to inform the teacher of his progress and diagnose his needs for further instruction. This is what is usually meant by "computer-managed instruction," or, as Brudner (1968) called it, "the computerized Man Friday." To attempt this sort of accounting it is not strictly necessary to have a computer; some projects of "individually prescribed instruction" have relied upon packets of hand-administered progress charts (see, for instance, the "learning packets" developed for the ES '70 program of the public schools of Philadelphia and other cities), but with computer record-keeping more elaborate schemes are possible. Perhaps the most ambitious proposal to explore this application of the computer was "Project PLAN," (Flanagan, 1967; Shanner, 1970; Gutkind, 1970), which envisioned daily input of evidence of progress. A trenchant comment on this point of view is contained in a cartoon that appeared in Educational Technology magazine (February 1969 issue, front cover): a child remarks to his companion: "I'm progressing at my own rate. Whose rate are you progressing at?"

In some CMI systems it is the teacher who sends the results of the various progress reports to the computer and receives the analyses, so that there is no direct interaction between student and machine. In others, the student may take tests at a terminal, or use punched cards or other machine-readable forms to record his own progress. In one way or another the computer is supplied frequent and detailed measures of the progress of individual students, and from these the teacher (or a computer program) can indicate the student's next assignment. The computer may take no direct part in the process of teaching. Its influence upon the content is primarily through the need for machine-readable or machine-analyzable tests. The computer is used to facilitate the work of the human instructor by taking over the detailed recording and analysis of individual progress.

2. Problem Solving. As part of his instruction, the student may use the computer to calculate answers to problems. In research installations, graduate schools, or even colleges, students may genuinely have problems of large size whose solution cannot otherwise be obtained. In their review of the use of computers in the teaching of physics, Schwarz, Kromhout, and Edwards (1969) remarked:

> Problem solving is generally accepted as a way of learning physics, but sometimes the calculations become

so difficult and time-consuming that the student loses sight of the physics involved. The efficiency and versatility of the computer can save student time and allow the assignment of more basic and more realistic problems.

At secondary or earlier levels, students rarely need to find solutions to problems involving large calculations. Rather, they are introduced to calculations in order to understand a process. The focus is on the nature of the problem or the manner in which it can be solved, rather than on the result itself. More than the authors of texts are usually aware, the procedures that they teach are derived from the assumed technology of computation. Bringing to a computer exercises developed for hand calculation may lose the point for which the exercises were constructed. If at the computer the students make use of "canned" programs whose logic is not evident to them, they may be able to produce right answers with little effort and equally little insight into a procedure. Whether or not that is desirable depends upon whether the emphasis of the lesson is upon the properties of the results or on properties of the procedure.

Effective use of the computer in "problem solving" has turned out to be more difficult than might at first have been supposed, and at present it seems fair to say that it has not yet had a major impact on the curriculum. A number of texts have appeared which contain computer exercises (for instance, the CAMP series for junior and senior high schools, which use the BASIC language, or the CRICISAM course in calculus, which uses flow-charts based upon a FORTRAN-like language). The persistent difficulty seems to be to integrate the computer exercises with the core presentation of the text, especially since the computer work seems to require style, notation, and procedures quite disjoint from those of the text. Having available a powerful computing device seems to require changes in the curriculum itself if effective use is to be made of it.

The approach described in the body of this report is perhaps more similar to "problem solving" than to any of the other established uses of computers in education. But what seems to us to be an important difference is that APL can be used not simply as a tool to carry out illustrative calculations, but as a way of stating and organizing the subject matter itself. Of course this has immediate benefits for the solution of problems, but it goes considerably beyond what is usually meant by "problem solving."

3. Simulation. The computer may be used to represent some aspects of a situation which the student would not otherwise

be able to experience. In teaching, simulation usually means giving the student information that he might have in a real situation, and permitting the experience to unfold in a plausible way as he interacts with it. In some simulations, the emphasis is on making concrete the effects of applying an abstraction (e.g. a rule or formula) to a specific situation. In others the emphasis is more upon the experience of filling a role that the student might otherwise find difficult to appreciate.

An example of the first sort of simulation --what might be called a "dry lab"-- is contained in the report by Stannard (1971). A physical or chemical apparatus is described, either in a manual or by a computer display. The student indicates from a terminal the actions he wants performed, and receives from the computer messages that indicate their effects. He can thus perform imaginary titrations more elaborately than might be possible in life, or experiments in dynamics with masses and speeds beyond anything he might reasonably hope to handle in actuality.

In the Sumerian game, as in other similar business and economic games used in management training, the student is invited to imagine that he has responsibilty for the ancient kingdom of Sumer (or some hypothetical corporation), and receives messages regarding affairs that call for his decisions. He communicates his decisions to the computer, and the program reacts to them.

In either of these types of simulation, the student cannot see and indeed is not expected to see how the simulator itself works. Rather, this "realistic" experience permits him to build up some impression of the phenomena involved, or perhaps provide the basis from which he can later derive some general principles.

It will be evident that the sort of open use of the computer that we have advocated in the body of this report differs from traditional simulation in two respects:

1. The extent to which the student can control the simulation: we have argued that a simulation may be constructed so that the students remains in control of the experiment, even when he is unable to see directly the inner structure by which it produces results.

2. The extent to which the simulation itself embodies the principles that the student is to discover: we have argued that the simulator can and should be written so that its own defintion, when finally revealed,

exemplifies the rules that the student must discover, and that in general the definitions of simulators should be both readable and comprehensible.

4. Teaching. The goal that has excited the greatest interest in educational research with computers is the goal of having the computer itself take over an important part of the process of teaching. The possibilities of large computer systems began to be realized at about the same time as the revival of interest in teaching machines and programmed instruction led by Skinner and his students. It appeared possible that the computer could present material to a student, elicit a response from him, evaluate his response, reward or correct him (as appropriate), and on the basis of his progress, decide what to present to him next. This approach combines the goal of tailoring the presentation to the individual needs and progress of the student with the behaviorist's hope of providing immediate reinforcement or immediate knowledge of results.

Just how closely the CAI movement is descended from programmed instruction has been a subject of some argument. Alpert and Bitzer (1970) called it "Misconception Number 1" to think of computer-based education and programmed instruction as synonymous. But that probably is not inconsistent with Shaplin's comment that

> [t]he stress [of CAI curriculum development] is on individual student progress, in logical, sequential steps, and on immediate reinforcement of learning. The direct lineage from the programmed-instruction movement and the reinforcement theory of learning is immediately apparent, with the computer program replacing the programmed text or teaching-machine program. In fact, the computer replaces the earlier concept of the teaching machine, providing infinitely more complex and promising possibilities. (Shaplin, 1967, page 41)

The possibility of a tutorial dialogue under computer supervision has led to interest among (at least) two rather different groups. There are those who see computer-managed learning and teaching as a vehicle for the study of the learning-teaching process itself, and there are those who simply want to use the computer as an alternative or even superior means to get the job done.

Educational psychologists have been intrigued by the possibility of at last obtaining highly controlled conditions under which to study the effects of schedules of reinforcement or other technical aspects of pedagogy. For instance,

Atkinson and Wilson (1968) report that a major focus of their CAI research at Stanford was "the development and testing of instructional strategies expressed as mathematical models" (page 76). Some have evidently conceived of computer-assisted instruction as a sort of simulation of the educational process.

But educational administrators, hard pressed by the expansion of enrollments, curricula, and costs, have seen the "computer tutor" as a way to deal with the backlog of ill-prepared students for whom it is especially difficult to get enough remedial instruction. The computer might provide patient attention to those whose progress is held up for lack of fundamental skills. At the same time, CAI offers the possibility of letting bright students proceed where they might otherwise be held back by the slower pace of the classroom. The goal of individualized instruction seems increasingly remote if we have to rely on having appropriately trained teachers in numbers large enough to achieve it solely by face-to-face human contacts.

Reformers from outside the educational establishment, such as those who led the "new math" and "new science" projects of the last decade, have also eyed CAI as a possible route to change in the curriculum. These have often been practitioners of the sciences in universities who became impatient with the training their students had received in the elementary and secondary schools, and sought to redirect its content and method. They found that it was one thing to state what they thought ought to be taught by the public schools, and quite another actually to bring about that change. That meant struggling with the large, diffuse, and not always sympathetic mass of the teaching profession, its professional schools, its governing boards, and even its publishers. At least some of these would-be rewriters of the curriculum must have imagined that a computer program over which they had complete control would be a more tractable medium and might offer them more leverage than they could command in the essentially political and sociological task of trying to change the practices of human teachers. Nevertheless, the more circumspect pioneers of this field maintained that the "computer tutor" or "conversational teaching machine" would not supplant the teacher, but rather would take over in a more efficient and more palatable way the drill in fundamental skills that is prerequisite to freeing both student and teacher for the more rewarding (or less programmable) aspects of their education.

It will be evident that the use of APL to represent concepts, and the open use of the computer that accompanies

it, are altogether different from tutorial CAI. We will return to the nature of this difference in the final pages of this appendix.

5. The Computer as the Subject of Study. Either because computers are becoming important to society, or more simply because there may be job opportunities around them, schools have often attempted to give their students some general familiarity with the nature of computing, or some direct experience in using a computer. "Computer Science and Technology," or more simply the use of data processing equipment, thus becomes an additional topic competing for emphasis in the school curriculum. So far it has had little impact upon either the content or the form of other courses.

In advanced secondary or early college courses on computing, it is common to provide the student a brief sketch of the history of computing, and then to introduce the device itself, with general descriptions of the functions of storage, input-output, and the arithmetic and logical units (see, for instance, the outline for high school programs on computing provided by Charp, 1967). Students may write a few simple programs in machine language, and then move to a symbolic assembler, or a compiler based upon FORTRAN or one of its variants. The flow chart is often used as a model of the computer approach to calculation. In some cases, flow-charting is presented as a sort of machine-free high level language (see, for instance, Stenberg & Walker, 1968).

In more business-oriented settings, students may have considerable experience in operating keypunches, verifiers, or tabulating equipment, or (more rarely) in the operation or programming of a larger machine. This sort of training is largely vocational, and often seems to lack application to the rest of the curriculum or even to the more general principles of computing.

By contrast, in chapter 3 of the main report we have undertaken to demonstrate that fairly general principles of computing can be stated, and operational computer models constructed by students, with little or no attention to the specifics of hardware, and that in this fashion general principles can be conveyed. More specific details can be added to the extent that they are deemed desirable, but without becoming the prerequisites for use of the computer.

Problems in the Further Development of the Computer's Roles

Surveying the roles that have thus far emerged for the computer in education, one is aware of certain problems that must be resolved before these roles can be developed more fully. As progress is made on one or another of the problems, the educational payoff from developing educational applications of computing (and hence the priorities of educational development) may alter correspondingly. It is in the perspective of these problems that we believe that open use of the computer, of the sort advocated in this report, provides an alternative that avoids several problems to which the traditional uses of computing are subject.

1. The Problem of Standardized Curriculum. Since the cardinal aim of most computer education projects has been the individualization of instruction, it may seem surprising to suggest that widespread use of computer-assisted or computer-managed instruction would bring with it strong pressures to make the curriculum more rigid and more standardized. Nevertheless, both computer-managed instruction (in which the computer does the micro-accounting but not the teaching) and computer-assisted instruction (in which the computer is responsible both for presenting material and for receiving and evaluating the student's response) rest upon the assumption of more or less standardized curricula, and especially the existence of standardized ways of measuring the student's progress through them.

To be sure, they offer the student an individualized route through the curriculum, so that no two individuals may receive exactly the same treatment. But the materials from which the student receives his individual selection, and, more important, the tests by which his progress is judged and his future work determined, must have been established in advance, and with sufficient breadth, certainty, and permanence to make it economic to build and operate the system that will individually pilot him through it. The individualization that these two approaches afford necessitates an underlying standardization (or at least codification) of goals, skills, and measures of performance.

Consultants on behavioral objectives will advise, and rightly, that in order to make this instruction effective its authors must very clearly identify the behavior that is to be produced, the criteria by which they will recognize it when it occurs, and the sequence in which it is to be built (see, for instance, Mager, 1962, or Walter, 1971). These behavioral inventories are intended to permit teachers to diagnose the

individual lacks of each student, and tailor instruction to his needs. But author and teacher should realize that once those behavioral objectives are clearly established, they are going to have to last for many different students if the project is to be economically viable.

By contrast, we are advocating a style of use in which the development task consists of preparing the conceptual framework and definitions. This framework is both itself easily modified, and within each framework of definitions there remains almost unlimited opportunity for variation in content or sequence of presentation.

2. The Problem of School Organization. The aim of individualized instruction, central to much work on CAI, requires that each student (a) have for a time sole use of a computer terminal, and (b) adjust the duration of his use to his invidivual level of achievement and pace of learning. These requirements conflict seriously with the traditional organization of schools into classes with more or less fixed membership and hours. Indeed, Cooley and Glaser (1969) remarked that

> ... it is virtually impossible to incorporate CAI into traditional schools where the classroom is the basis for instructional decisions and scheduling. (page 582)

It may well be that the schools need to break out of the rigidity of their traditional organization, and indeed many of the innovations in school procedure have been in the direction of greater flexibility for the individual. Nevertheless, a role for the computer which cannot be adapted to use by a group (be it a class of fixed size, or an ad hoc discussion group) seems overly restrictive. An underlying issue is whether the computer is seen as a means of individualized presentation, or whether it also provides material or ideas that can become the subject of class discussion or activity. We have argued that APL provides a common language, executable at the terminal in the same form that it would be written in texts or at the blackboard. This means not only that computing work is facilitated, but also that it is easy to integrate computing work with the assignments and procedures of the more conventional classroom. Indeed, it is possible to take advantage of the conceptual benefits of expressing concepts in APL even if there is no use of computer terminals.

3. The Problem of Costs. Parallel to the problem of school organization is the problem of associated costs. The individualization of instruction requires a corresponding individualization of the interface with the computer: enough

time or enough terminals are required for all the instruction that the computer is to give, for all the students who are to receive it. Although the cost of terminals has tended to decrease and in the long run is likely to decrease further, a large number of terminal-hours per student can still make for very high operating costs of a computer educational system.

If the computer is actually to initiate and administer the instruction of individuals (as is widely assumed in CAI programs), there is also a major cost in the development of the programs themselves. This cost may be spread over all the users of the program over all the years it is offered, but still is large in absolute terms.

Even inexpensive systems of presentation or computation may still be wasteful if they do not produce a valued educational result. In the final analysis any system must be judged by the value of its educational effect per dollar spent, rather than the number of terminal hours or cpu seconds purchased. This seems to us a most crucial point: the installation costs of systems such as APL is large, but, we believe, is justified by the large payoff in educational insights for small amounts of computation. A critical factor in establishing such an educational use of the computer, however, is to establish a wide enough community of users over which to share the fixed costs of central equipment. Where computing is offered to a diverse group of users by an installation acting as a "computer utility" the marginal cost to an educational institution may be greatly reduced.

4. The Problem of Teacher Preparation. The more the use of the computer is individualized, the greater the conflict with traditional school organization. But the more the use of the computer is absorbed into the conventional classroom, the greater the burden on the teacher to assimilate its concepts and its use into his teaching. As long as the computer is a separate device to which students go privately and individually, the teacher may have little need to involve himself in what it does. The harassed school administrator may find this an appealing element in the prospectus for a CAI system; the teacher may be relieved rather than disturbed that some of his previous functions may be supplanted by a machine. He can rely on the authors of CAI programs to decide how well his students are doing and what they need next. By contrast, when the computer is introduced into a classsroom activity which is led not by the computer but by the teacher, the teacher has the new burden of relating computer work to the rest of his instruction. This new aid, if he makes much use of it, may seriously disrupt the sequence of his course, or make obsolete his carefully built-up store of problems and

examples. Instead of feeling further ahead in his work of teaching, he may find himself in serious need of additional training, but with small opportunity to obtain it. This seems to us the most serious hurdle to ready adoption of APL in the classroom: it requires the informed and active participation of teachers. By contrast, CAI lessons offer the prospect that they can be "plugged in" and run in the classrooms of teachers who know nothing at all about how the lessons were prepared or what it is they teach. There remains a need (today largely unmet) to provide orientation and training for teachers who make use of computing, and especially to integrate such training with the rest of their preparation.

5. The Problem of Natural Language. Under each of the alternative computer roles, there is some sort of communication between the student and a machine. If the machine is used only for accounting, the communication may be limited to the letters or numbers that represent his grades, or his scores on tests, or his name and other identifying facts. Already students are (sometimes painfully) aware of the impersonality of the codes and ID numbers by which this communication is commonly achieved.

More sophisticated systems may address the student in more human terms. Indeed, designers of conversational systems seem to bend over backwards to have their systems emit colloquial remarks. But it remains unlikely that a computer will soon be able to understand a reasonable but unusual inquiry made in natural language. The plausible verbosity of systems such as ELIZA (Weizenbaum, 1966) or the "conversational teaching machine" reported by Feurzeig (1964) or by Swets and Feurzeig (1965) conceal the fact that those systems can not understand even the simplest English sentence, but must rely on being able to identify within the student's responses particular words that it has been warned to expect.

The authors of CAI lessons often need to address the student in terms already familiar to him, and hence to attempt to use natural language. This is usually done by storing in advance the various blocks of text that the program will cause to be printed. The program is able to accept something approaching natural-language response from the student only by posing multiple choice questions to which the student makes a simple response such as pointing with a light-pen. Even when the program invites the student to enter a "constructed response," its analysis is limited to searching his brief reply for key words or phrases, or statistical approximations of them. Syntactic analyzers and dictionaries sophisticated enough and large enough to deal effectively with the likely range even of rather constrained student responses are

difficult to build, and cannot yet be readily borrowed from other projects. The result is that at present tutorial systems can emit prepared statements in natural language, but in general can neither understand natural input nor construct appropriate new natural-language replies. No doubt this is an area in which future research will expand present capabilities. But for the present, it is difficult if not impossible to construct a computer system that can respond to a student's unanticipated question, and hence impossible for the computer to provide one of the crucial aspects of the teacher's role.

6. The Problem of Computer Language. The difficulties of natural-languange communication with the computer may be avoided if the role of tutor is abandoned, and instead the emphasis is placed upon the computer as a problem-solving tool at the student's disposal. This usually means that the student will control what work the computer does on his behalf by writing instructions or programs that the computer is to execute. But with this shift in roles and goals, the question of the language of communication between student and machine arises in another way.

Over the last ten years, the development of time-sharing and of mini-computers (many of which are larger than the "full size" machines of a decade ago) has made the computer more physically accessible. The development of high level languages and conversational systems has made it easier to write a program and debug it. But it frequently remains the case that the student must learn a great deal about the computer, its operating system and its language before he can do serious work. If what he must learn about the computer itself is irrelevant to his subject of study, it must be regarded as an educational overhead of using the machine. How high is this cost compared to the benefits of computing? In 1967 Feurzeig claimed that

> New languages have made computers so easy to use that elementary school pupils are now formulating and solving mathematical problems through the use of computer terminals located in school classrooms. (p. 86)

Nevertheless, the spread of direct use of computers by students has been slow. Perhaps not everyone has had access to a simple system. The concept of "simplicity" in computer systems is treacherous: if it turns out to mean that the computer is confined to simple tasks, the apparent ease of initial use may disguise difficulties that will dog the user who wants to go on to something more interesting or more relevant.

Our central claim for APL is that it provides simultaneously the simplicity that makes it easily acquired and used by students, and the power to deal effectively with a wide range of applications. In particular, it seems to us almost uniquely open-ended, in that the student who is introduced to APL in the early years of secondary school can expect both to make direct use of it immediately and to be able to continue to develop his use of the language as he proceeds to more advanced stages of his education, up to the most advanced professional levels.

7. The Problem of Relevance. Does what the student learns at the computer really relate to what we want him to get out of his education? If the computer is used in a tutorial role, the problem is whether the authors can indeed fit what needs to be taught into the frames or response units that make up the program, and can respond appropriately to the student's needs. But if the computer is used for problem solving, or as the subject of study, we are directly challenged to make the student's experience with the computer relevant to his education. If a student is studying the computer to learn some immediately marketable skills, that may be justification enough, at least in vocational training-- although even there what he learns may be in chronic danger of obsolesence. But if we want more from the computer than vocational skills, how are concepts derived from computing, or experiences gained while computing, related to the rest of human knowledge? What are the underlying fundamental principles that may be illustrated with computing?

We state the question in that way because we believe that such fundamental principles do exist, and that, given an appropriate approach, they can indeed illuminate the understanding of many topics. Unfortunately, in practice these principles are frequently obscured by the technical details that surround use of a computer. Sometimes these irrelevant exigencies of computing, rather than being avoided or eliminated, are instead mistaken for its fundamental ideas. For instance the School Mathematics Study Group's unit on computing, circulated in 1966, spoke of an "algorithmic approach" and seemed to mean by that the reduction of a function's definition to a sequence of very small steps. We suggest that the need to divide the definition may have provided more insight into the limitations of the machine than into the nature of the function thus divided. We believe that APL may be used in a way that is maximally relevant to the topics to be taught because the language is minimally constrained by the computer that executes it.

What Does Computer Aided Instruction Aid: Process or Content?

In order to produce programmed material, or to measure progress (essential elements of CAI and CMI projects) it is necessary to have defined measurable objectives. The act of formulating such explicit objectives may indirectly provoke revision of the content of instruction, especially by forcing authors to avoid topics for which they have ill-defined or unquantifiable goals. The ability of the computer to handle large calculations has another sort of effect upon content, since it often permits the study of larger or more realistic problems in science or engineering. Nevertheless, neither of these effects would usually be considered a new way of thinking about the content of a course.

Reviewing the five earlier roles that emerged for the computer in education, it is striking that they contributed primarily to the process of instruction, and relatively little to the content. Individualization, the key goal of much CAI work, is explicitly aimed at instructional process, by permitting control of reinforcement, pace, and the selection of routes through the material. In principle, CAI methods of presentation can be adapted to any content, and are largely independent of what that content is. CAI could be used equally as well to present, drill, and test the retention of nonsense syllables as any other body of content. The same cannot be said of the use of APL that we have been advocating: that use is dependent upon concise and effective statement of the inner structure by which a topic is organized. In the absence of such an inner structure, the method is inappropriate. But of course most disciplines do indeed have such a structure. Our use of the computer thus focusses attention on ways in which that structure can be represented and made tangible.

A Question of Philosophy and Style

Underlying many of the contrasts between the open use of APL and the various more traditional uses of computing in education there seems to us to be an issue of educational philosophy: to what extent is the student to be free to pose his own questions, make his own discoveries, formulate his own answers? It seems to us that much of what has gone before has employed the computer according to what is at base an authoritarian, paternalistic conception of education. It stems from the view that there is no fundamental difference between education and training-- each consists of the acquisition of behavior, and the task of education, like that of training, is to enumerate the component skills that are

needed, induce students to produce the requisite behavior, and then verify that it has been learned. The teacher (or course author, or computer) knows best, and will (with individualized care and inhuman patience) make sure that the student learns what wiser heads than his have decided he should learn. Whatever the merit or effectiveness of such an approach to education, it is by no means the only style of schooling that can be assisted by computing. It is our belief that a spirit and style of open inquiry, in which the computer is the student's tool rather than his mentor, is possible with APL, and is more in keeping with a humane conception of the nature of science and the goal of teaching.

Acknowledgments

This paper represents the authors' efforts to develop and apply a conception of the nature and place of APL in educational computing which they learned from Adin Falkoff and Ken Iverson. Throughout the formative period of this work in Philadelphia, the authors benefitted from many discussions with them. In addition, they acknowledge the guidance provided by Professor Aldo Romano, scientific director of the Computer-Based Learning Systems Laboratory of C.S.A.T.A., oriented to a problem solving approach to the teaching of science, and by Dr Attilio Stajano, director of the IBM Scientific Center at Bari. Dr Howard Peele of the University of Massachusetts made many useful comments on the manuscript, and his seminar at Hampshire College provided a forum for discussion of an early version of some of the material. The discussions of the use of APL in the classroom draw on the experience of Miss Linda Alvord, Scotch Plains-Fanwood High School, Scotch Plains, New Jersey, Mr Nat Bates, the Belmont Hill School, Belmont, Massachusetts, and Mr John Brown, formerly at Lower Canada College and now at Dawson College, Montreal. Our thanks to all these people. Papers based upon part of this material were presented to the Conference on APL sponsored by l'Institut de Recherche d'Informatique et d'Automatique in Paris during September, 1971, and to the International Computing Symposium sponsored by the European chapters of the Association for Computing Machinery in Venice during April 1972.

References

Allen, D W. Computer-built Schedules and Educational Innovation. in: Bushnell & Allen, 1967, pp. 51-58.

Alpert, D, & D L Bitzer. Advances in Computer-based Education. Science, 1970, 167, 1582-1590.

Atkinson, R C, & H A Wilson. Computer-Assisted Instruction. Science, 1968, 162, 73-77.

Berry, P C, A D Falkoff & K E Iverson. Using the Computer to Compute: a Direct but Neglected Approach to Teaching Mathematics. IBM Corporation: Philadelphia Scientific Center, TR. 320-2988 May 1970. Also appears in Education 70, Proceedings of the IFIP World Conference on Computer Education, Amsterdam, 1970.

Bartoli, G, C Dell'Aquila, & A Romano. A Conversational Interactive System in the Teaching of Physics. International Computing Symposium, Venice, Italy, April 13, 1972.

Brudner, H J. Computer-Managed Instruction. Science, 1968, 162, 970-976.

Bushnell, D D, & D W Allen. The Computer in American Education. New York: Wiley, 1967.

Charp, Sylvia. Computer Programming Courses in Secondary Schools. in: Bushnell & Allen, 1967, pp. 137-155.

Cooley, W W, & R Glaser. The Computer and Individualized Instruction. Science, 1969, 166, 574-582.

Falkoff, A D. Algorithms for Parallel-Search Memories. Journal of the Association for Computing Machinery, 1962, 9, 488-511.

Falkoff, A D. Designing a General Purpose Language, pp. 234-251 in High Level Languages, International State of the Art Report, INFOTECH, Maidenhead, Berks., England, 1972.

Falkoff, A D, K E Iverson & E H Sussenguth. A Formal Description of System/360. IBM Systems Journal, 1964, vol. 3, #3.

Feurzeig, W. New Instructional Potentials of Information Technology. IEEE Transactions on Human Factors in Electronics, 1967, 8, 84-88.

Feurzeig, W. Conversational Teaching Machine. Datamation, June, 1964.

Flanagan, J C. Project PLAN: A program of individualized planning and individualized instruction. Project ARISTOTLE Symposium, National Security Industrial Association, Washington, D.C. 1967.

Gardner, M. Mathematical Games. Scientific American, October 1970.

Gutkind, L A. Probing Project PLAN. Scholastic Teacher, 5 October 1970.

Hagamen, W D, J C Weber, D J Linden, & S S Murphy. A Tutorial System (ATS) Concepts and Facilities Manual, preliminary version. New York: Department of Anatomy, Cornell University Medical College, 1971.

Hellerman, H. Digital Computer System Principles. New York: McGraw-Hill, 1967.

Instructional Technology Committee. Educational Technology in Higher Education. Washington: National Academy of Engineering, 1969.

Iverson, K E. A Programming Language. New York: Wiley, 1962.

Iverson, K E. Algebra: An Algorithmic Treatment, Menlo Park, California: Addison-Wesley, 1972.

Iverson, K E. The Use of APL in Teaching. IBM Corporation, TR 320-0996, May 1969.

Leonard, J. The Design of a Computer-Based Educational Game. In M Inbar & C S Stoll, Social Science Simulations. New York: Free Press, 1971.

Macauley, T. CAL/APL: Computer Assisted Learning Author's Manual. Costa Mesa, California: Orange Coast Junior College District, 1969.

Mager, R. Preparing Instructional Objectives. Palo Alto: Fearon, 1962.

Penfield, P, Jr. MARTHA. Cambridge: The MIT Press, 1971.

Romano, A. An Instructional Interactive Model for Science Teaching. Proceedings of the International Summer School on the Computer in Education, Pugnochiuso, Italy, 1972. In press.

Schwarz, G, Ora M Kromhout & S Edwards. Computers in Physics Instruction. Physics Today, Sept. 1969, 41-49.

Shanner, W M. A System of Individualized Instruction Utilizing Currently Available Instructional Materials. New York: Westinghouse Learning Corp., 1970.

Shaplin, J T. Computer-based Instruction and Curriculum Reform. in: Bushnell & Allen, 1967, pp. 36-43.

Spence, R. Resistive Circuits. IBM Corporation: Philadelphia Scientific Center, TR. 320-3018, March 1973.

Stannard, C R. Use of Digital Computer in Introductory General Physics. IBM Corporation, GC20-1746, 1972.

Stenberg, W, & R J Walker. Calculus: a Computer-Oriented Presentation. CRICISAM (Center for Research in College Instruction in Science and Mathematics), University of Florida, Tallahassee, 1968.

Suppes, P & Mona Morningstar. Computer Assisted Instruction. Science, 1969, 166, 343-350.

Suppes, P, M Jerman & D Brian. Computer Assisted Instruction: the 1965-66 Stanford Arithmetic Program. New York: Academic Press, 1968.

Swets, J A, & W Feurzeig. Computer-Aided Instruction. Science, 1965, 150, 572-576.

Walter, K A. Authoring Individualized Learning Modules: a Teacher's Training Manual. Kensington, Maryland: Montgomery County Public Schools.

Weizenbaum, J. ELIZA--A Computer Program for the Study of Natural Language Communication Between Man and Machine. Communications of the ACM, 1966, 9, 36-43.

Wertheimer, Max. Productive Thinking. First edition, 1954; enlarged edition, New York: Harpers, 1959.

APL PRESS
Pleasantville
New York